pyanp Documentation

Release 0.3

Dr. Bill Adams

Apr 28, 2020

Contents:

1 Programmers Reference PyANP **1**

1.1 AHPTree class . . . 1
1.2 Group pairwise comparison . . . 5
1.3 Priority calculations module . . . 8
1.4 Limit matrix calculations . . . 11
1.5 Example matrices module . . . 14
1.6 General functions module . . . 15
1.7 Direct data class . . . 16
1.8 Prioritizer class . . . 17
1.9 ANP Row Sensitivity . . . 18
1.10 Rating class . . . 23
1.11 ANP structure module . . . 26

2 Indices and tables **35**

Python Module Index **37**

Index **39**

CHAPTER 1

Programmers Reference PyANP

1.1 AHPTree class

class pyanp.ahptree.**AHPTree**(*root_name='Goal'*, *alt_names=None*)
Represents all of the data of an ahp tree.

__init__(*root_name='Goal'*, *alt_names=None*)
Creates a new AHPTree object

Parameters

- **root_name** – The name of the root node of the tree, defaults to Goal.
- **alt_names** – The alts to start this tree with.

add_alt(*alt_name: str*) → None
Adds an alternative to this tree and all of the nodes in the tree.

Parameters **alt_name** – The name of the new alternative to add.

Returns Nothing

Raises **ValueError** – If an alternative already existed with the given name

add_child(*childname: str*, *undername: str = None*) → None
Adds a child node of a given name under a node.

Parameters

- **childname** – The name of the child to add.
- **undername** – The name of the node to add the child under

Returns Nothing

Raises **ValueError** – If undername was not a node, or if childname already existed as a node.

add_user(*user: str*) → None
Adds the user to this AHPTree object. :param user: The name of the user to add :return: Nothing :raises ValueError: If the user already existed.

alt_direct(*wrt: str*, *alt_name: str*, *val: float*) → None
Directly sets the alternative score under wrt node. See AHPTreeNode.alt_direct for more information as that is the function that does the hard work. :param wrt: The name of the wrt node. :param alt_name: The name of the alternative to direclty set. :param val: The new directly set value. :return: Nothing :raises ValueError: * If there is no alternative by that name * If the wrt node did not exist

alt_incon_std(*username*, *wrt: str = None*) → float
Calcualtes the standard inconsistency score for the pairwise comparison of the alts for the given user

Parameters

- **username** – The string name/names of users to do the inconsistency for. If more than one user we average their pairwise comparison matrices and then calculate the incosnsitency of the result.
- **wrt** – The name of the node to get the inconsistency around. If None, we use the root node.

Returns The standard Saaty inconsistency score.

alt_pwmatrix(*username*, *wrt: str*) → numpy.ndarray
Gets the alternative pairwise comparison matrix for the alts under wrt, assuming the wrt node has the alternatives under it and they are pairwise compared.

Parameters

- **username** – The name/names of the users to get the pairwise comparison of.
- **wrt** – The name of the wrt node, or the AHPTreeNode object.

Returns A numpy array of the pairwise comparison information. If more than one user specified in usernames param we take the average of the group.

altpw(*username: str*, *wrt: str*, *row: str*, *col: str*, *val*, *createUnknownUser=True*) → None
Pairwise compares a alts for a given user.

Parameters **username** – The name of the user to do the comparison for. If the user doesn't exist, this will create

the user if createUnknownUser is True, otherwise it will raise an exception

Parameters

- **wrt** – The name of the wrt node.
- **row** – The name of the row alt for the comparison, i.e. the dominant node.
- **col** – The name of the column alt for the comparison, i.e. the recessive node.
- **val** – The vote value

Returns Nothing

Raises **ValueError** – If wrt, row, or col node did not exist. Also if username did not exist and

createUnknownUsers is False.

get_node(*nodename: str*) → pyanp.ahptree.AHPTreeNode

Parameters **nodename** – The string name of the node to get. If None, we return the root node.

If nodename is actually an AHPTreeObject, we simply return that object.

Returns The AHPTreeNode object corresponding to the node with the given name

Raises **KeyError** – If no such node existed

get_nodes_hash() → dict

Returns

A dictionary of nodeName:nodeObject for all nodes in this tree.

global_priority(*username=None*, *rvalSeries=None*, *undername: str = None*, *parentMultiplier=1.0*) → pandas.core.series.Series
Calculates and returns the global priorities of the nodes.

Parameters

- **username** – The name/names of the users to calculate for. None means the group average.
- **rvalSeries** – If not None, add the results to that series
- **undername** – If None, use the root node, otherwise a string for the name of the node to go under. Internally we also allow for AHPTreeNode's to be passed in this way.
- **parentMultiplier** – The value to multiply the child priorities by.

Returns The global priorities as a Series whose index is the node names, and values are the global priorities.

global_priority_table() → pandas.core.frame.DataFrame
Calculates the global priorities for every user, and the group

Returns A dataframe whose columns are "Group" for the total group average, and then each user name. The rows are the node names, and values are the global priority for the given node and user.

incon_std(*username*, *wrt: str = None*) → float
Calcualtes the standard inconsistency score for the pairwise comparison of the children nodes for the given user

Parameters

- **username** – The string name/names of users to do the inconsistency for. If more than one user we average their pairwise comparison matrices and then calculate the incosnsitency of the result.
- **wrt** – The name of the node to get the inconsistency around. If None, we use the root node.

Returns The standard Saaty inconsistency score.

incon_std_series(*username: str*) → pandas.core.series.Series
Calculates the inconsistency for all wrt nodes for a user / user group. See AHPTree.incon_std() for details about the actual calculation.

Parameters **username** – The name/names of the user to calculate the inconsistency for.

Returns A pandas.Series whose index is wrt node names, and whose values are the inconsistency of the given user(s) on that comparison.

incond_std_table() → pandas.core.frame.DataFrame
Calculates the inconsistency for all users and wrt nodes in this tree.

Returns A pandas.DataFrame whose columns are users (first column is called "Group" and is for the group average) and whose rows are wrt nodes. The values are the inconsistencies for the given user on the give wrt node's pairwise comparison.

isalt(*name: str*) → bool
Tells if the given alternative name is an alternative in this tree. :param name: The name of the alternative to check. :return: True if the alternative is in the list of alts for this tree, false otherwise.

nalts()

Returns

The number of alternatives in this tree.

node_pwmatrix(*username*, *wrt: str*) → numpy.ndarray
Gets the pairwise comparison matrix for the nodes under wrt.

Parameters

- **username** – The name/names of the users to get the pairwise comparison of.
- **wrt** – The name of the wrt node, or the AHPTreeNode object.

Returns A numpy array of the pairwise comparison information. If more than one user specified in usernames param we take the average of the group.

nodenames(*undername: str = None*, *rval=None*) → list
Name of all nodes under the given node, including that node.

Parameters **undername** – The name of the node to get all nodes under, but only if underNode is not set.

It can also be an AHPTreeNode, but that is really for internal use only

Parameters **rval** – If not

Returns The node names as a list

nodepw(*username: str*, *wrt: str*, *row: str*, *col: str*, *val*, *createUnknownUser=True*) → None
Pairwise compares a nodes for a given user.

Parameters **username** – The name of the user to do the comparison for. If the user doesn't exist, this will create

the user if createUnknownUser is True, otherwise it will raise an exception

Parameters

- **wrt** – The name of the wrt node.
- **row** – The name of the row node for the comparison, i.e. the dominant node.
- **col** – The name of the column node for the comparison, i.e. the recessive node.
- **val** – The vote value

Returns Nothing

Raises **ValueError** – If wrt, row, or col node did not exist. Also if username did not exist and

createUnknownUsers is False.

nodes(*undername: str = None*, *rval=None*)
Returns the AHPTreeNode objects under the given node, including that node

Parameters

- **undername** – The string name of the node to get the nodes under. It can also be an AHPTreeNode object as well. If None it means the root node.
- **rval** – If not None, it should be a list to add the AHPTreeNode's to.

Returns The list of AHPTreeNode objects under the given node.

priority(*username=None*, *ptype: pyanp.prioritizer.PriorityType = None*) → pandas.core.series.Series

Calculates the scores of the alternatives. Calls AHPTree.synthesize() first to calculate. :param username: The name (or list of names) of the user (users) to synthensize. If username is None, we calculate for the group. :param ptype: Do we want to rescale the priorities to add to 1 (normalize), or so that the largest value is a 1 (idealize), or just leave them unscaled (Raw). :return: The alternative scores, which is a pd.Series whose index is alternative names, and values are the scores.

priority_table() → pandas.core.frame.DataFrame

Returns A dataframe whose columns are "Group" for the total group average, and then each user name.

The rows are the alternative names, and the values are the alternative scores for each user.

synthesize(*username=None*) → None

Does ahp tree synthesis to calculate the alternative scores wrt to all nodes in the tree.

Parameters **username** – The name/names of the user/users to synthesize wrt. If None, that means do the full group average.

Returns

Nothing

usernames() → list

Returns

The names of the users in this tree.

1.2 Group pairwise comparison

Group pairwise object and calculations. See *pyanp.priority* for all methods of calculating priorities from a pairwise comparison matrix in addition to inconsistency calculations.

class pyanp.pairwise.**Pairwise**(*alts=None*, *users=None*, *demographic_cols=None*)

Creates a new group pairwise comparison object.

Parameters

- **alts** – The list alternatives (things you are comparing) to start with. Should be a list-like object of strings.
- **users** – The users to start the group pairwise comparison object with. It should be a list-like object of strings.
- **demographic_cols** – The names of the demographic columns to start the group pairwise comparison object with. It should be a list-like object of strings.

add_alt(*alt_name: str*, *ignore_existing=False*) → None

Adds an alternative (thing you are pairwise comparing) to this group pairwise comparison object.

Parameters **alt_name** – The name of the alternative to add.

Returns Nothing

Raises **ValueError** – If the alternative already esisted.

add_user(*user_name: str*) → None

Adds a user to this group pairwise comparison object.

Parameters **user_name** – The name of the user to add

Returns Nothing

Raises **ValueError** – If the user already existed.

alt_index(*alt_name_or_index*) → int
Find the index (integer location) of the given alternative in the pairwise comparison matrices.

Parameters **alt_name_or_index** – If this is an integer, we simply return that integer. Otherwise we look up the index of the alternative name in the list of alternatives in this object.

Returns The index that alternative has in the pairwise comparison matrices.

alt_names()

Returns List of string alt names

data_names(*append_to=None*, *post_pend=''*)

incon_std(*user_name=None*) → float
Calculates the standard Saaty pairwise comparison inconsistency for a user or group of users.

Parameters **user_name** – The name/names of the users to get the inconsistency of. If None, we get the inconsistency of the group average matrix. If it is a string, we get the inconsistency of that user. If it is a list of users, we get the inconsistency of the group average for that list of users.

Returns The Saaty inconsistency score.

is_alt(*alt_name: str*) → bool
Checks if an alternative (a thing you are pairwise comparing) exists in this group pairwise comparison object.

Parameters **alt_name** – The name of the alternative to check for.

Returns True/False

is_user(*user_name: str*) → bool
Checks if a user exists in this group pairwise comparison object.

Parameters **user_name** – The name of the user to look for

Returns True/False

matrix(*user_name=None*, *createUnknownUser: bool = True*, *as_df=False*) → numpy.ndarray
Gets the pairwise comparison for a user or group of users.

Parameters

- **user_name** – The name/names of the user/users to get the comparisons of. If None, that means to get the group average for all users. If it is a string, that means get the pairwise comparison matrix of that user. If it is a list-like of strings, we get the group average matrix for all of those users.
- **createUnknownUser** – If True and the user_name did not exist, we should create that user. Otherwise throw an error if we request for a non-existant user.
- **as_df** – If True return as pandas.DataFrame with index/column names as the alt names, otherwise return numpy.ndarray

Returns The numpy array of the pairwise comparisons, or DataFrame if as_df is True

Raises **ValueError** – If createUnknownUser=False and we request for a single non-existant user.

nalts() → int

Returns The number of alternatives (things you are pairwise comparing) in this group pairwise comparison object.

priority(*username=None*, *ptype: pyanp.prioritizer.PriorityType = None*)
Calculates the resulting priority for the given user / users.

Parameters

- **user_name** – The name/names of the users to calculate the priority of. If None, we get the priority of the group average matrix. If it is a string, we get the priority of that user. If it is a list of users, we get the priority of the group average for that list of users.
- **ptype** – How should we normalize the resulting priorities (if at all).

Returns A pandas.Series whose indices are the alternative names and whose values are the priorities of those alternatives.

unvote(*user_name: str*, *row*, *col*, *createUnknownUser: bool = True*) → None
Unsets a pairwise comparison

Parameters

- **user_name** – The string name of the user whose pairwise comparison vote you wish to unset.
- **row** – The integer or string name of the row to compare at.
- **col** – The integer or string name of the column to compare at.
- **createUnknownUser** – If True and user_name does not exist in this object, we will create it first, then do the unset operation. Otherwise it throws an exception for unknown users.

Returns Nothing

Raises **ValueError** – If the user does not exist and createUnknownUsers is False.

usernames()

Returns A list of the users in this group pairwise comparison object.

vote(*user_name: str*, *row*, *col*, *val: float = 0*, *createUnknownUser: bool = True*) → None
Changes a single pairwise value for a single user.

Parameters

- **user_name** – The string name of the user whose pairwise comparison vote you wish to change.
- **row** – The integer or string name of the row to compare at.
- **col** – The integer or string name of the column to compare at.
- **val** – The new pairwise comparison value
- **createUnknownUser** – If True and user_name does not exist in this object, we will create it first, then do the comparison. Otherwise it throws an exception for unknown users.

Returns Nothing

Raises **ValueError** – If the user does not exist and createUnknownUsers is False.

vote_matrix(*user_name: str*, *val=<class 'numpy.ndarray'>*, *createUnknownUser: bool = True*)
Sets the vote matrix for a user

Parameters

- **user_name** –
- **val** –

Returns

vote_series(*votes: pandas.core.series.Series*, *row*, *col*, *createUnknownUser: bool = True*) → None
Changes a single pairwise value for a series of users.

Parameters

- **votes** – Series whose index is usernames and values are their votes.
- **row** – The integer or string name of the row to compare at.
- **col** – The integer or string name of the column to compare at.
- **createUnknownUser** – If True and a username does not exist in this object, we will create it first, then do the comparison. Otherwise it throws an exception for unknown users.

Returns Nothing

Raises **ValueError** – If the user does not exist and createUnknownUsers is False.

pyanp.pairwise.**add_place**(*mat*)
Adds a row and column to the end of a matrix, and makes the last entry 1, rest of the added entries are zeroes

Parameters **mat** – The matrix to add an entry to.

Returns New matrix

pyanp.pairwise.**geom_avg_mats**(*mats*) → numpy.ndarray
Calculates the geometric average of the given matrices.

Parameters **mats** – A list-like object of numpy arrays

Returns A numpy array that is the geometric average

1.3 Priority calculations module

All pairwise matrix to priority vector calculations

@author: Dr. Bill Adams

pyanp.priority.**geom_avg**(*vals*) → float
Compute the geometric average of a list of values.

Parameters **vals** – A list-like of numbers

Returns The geometric average of the values.

pyanp.priority.**geom_avg_mat**(*mat*, *coeffs=None*) → numpy.ndarray
Computes the geometric average of the columns of a matrix.

Parameters

- **mat** – Must be an numpy.array of shape [nRows, nCols]
- **coeffs** – If not None, it is a list like object with nColsOfMat elements. We multiply column 0 of mat by coeffs[0], column 1 of mat by coeffs[1], etc and then do the geometric average of the columns. Essentially this weights the columns.

Returns An np.array of dimension [nRowsOfMat], i.e. a vector. that is the weighted geometric average of the columns of the matrix mat.

pyanp.priority.**harker_fix**(*mat: numpy.ndarray*) → numpy.ndarray

Performs Harker's fix on the numpy matrix mat. It returns a copy with the fix. The function does not change the matrix mat.

Parameters **mat** – A square numpy.

Returns A copy of mat with Harker's fix applied to it

pyanp.priority.**incon_gci**(*mat: numpy.ndarray*, *use_harker: bool = True*, *only_count_nonzero=True*) → float

Calculates the inconsistency of a pairwise matrix using the GCI formula from

Parameters

- **mat** – A numpy.array of shape [size,size] of pairwise comparisons.
- **use_harker** – Should we apply Harker's fix before the calculation?
- **only_count_nonzero** – Should we only average nonzero comparisons (this is not the official defn of gci)

Returns The inconsistency.

pyanp.priority.**incon_std**(*mat: numpy.ndarray*, *error: float = 1e-10*, *use_harker: bool = True*) → float

Calculates the inconsistency of a pairwise matrix using the standard Saaty AHP/ANP theoretic formula.

Parameters

- **mat** – A numpy.array of shape [size,size] of pairwise comparisons.
- **error** – The error to use for the pri_eigen calculation
- **use_harker** – Should we apply Harker's fix before the calculation?

Returns The inconsistency.

pyanp.priority.**inconsistency_divisor**(*mat_or_size*) → float

Calculates the inconsistency divisor for a matrix, or the size of a matrix. The inconsistency divisor is what you divide (eigenvalue - size) by to get the inconsistency.

Parameters **mat_or_size** – Either a pairwise matrix, or simply the size of the pairwise matrix (which is what determines the inconsistency divisor).

Returns The inconsistency divisor

pyanp.priority.**mat_gci**(*mat1: numpy.ndarray*, *mat2: numpy.ndarray*, *only_count_nonzero=True*) → float

Calculates the GCI of two matrices :param mat1: :param mat2: :return:

pyanp.priority.**prerr_euclidratio**(*pwmat*, *privec*)

Calculates the euclidean distance error between the pairwise matrix and the ratio matrix of a priority vector.

This calculates using the following formula

$$\sqrt{\sum_{i,j}\left(\frac{pwmat[i,j]-privec[i]}{privec[j]}\right)^2}$$

Parameters

- **pwmat** – A numpy.array of shape [size, size] of pairwise comparisons.

- **privec** – A numpy.array of share [size] of the priority vector to compare this pairwise matrix to.

Returns The error/distance between the ratio matrix and the matrix.

pyanp.priority.**prerr_ratio_avg**(*pwmat*, *privec*)

Calculates priority error using the arithmetic average of ratio distance of pwmat from the ratio matrix of privec

It averages:

$$ratio\_greater\_1(pwmat[i,j],(privec[i]/privec[j]))-1$$

where

$$ratio\_greater\_1(a,b)=\begin{cases}1 & \text{if a or b = 0}\\ max(a/b,b/a) & \text{otherwise}\end{cases}.$$

Parameters

- **pwmat** – A numpy.array of shape [size, size] of pairwise comparisons
- **privec** – A numpy.array of shape [size] of priortiy vector

Returns The ratio average priority vector

pyanp.priority.**prerr_ratio_prod**(*pwmat*, *privec*)

Calculates priority error using the geometric average of ratios of pwmat and the ratio matrix of privec, the formula is:

$$\sqrt[n(n-1)/2]{\prod_{i=1,j=i+1} ratio\_greater\_1(pwmat[i,j],privec[i]/privec[j])}$$

Parameters

- **pwmat** – A numpy.array of shape [size, size] of pairwise comparisons
- **privec** – A numpy.array of shape [size] of priortiy vector

Returns The calculated error

pyanp.priority.**pri_eigen**(*mat: numpy.ndarray*, *error: float = 1e-10*, *use_harker: bool = False*, *return_eigenval: bool = False*)

Calculates the largest eigen vector of a matrix.

Parameters

- **mat** – A square numpy array.
- **use_harker=False** – Should we apply Harker's fix before computing?
- **return_eigenval=False** – If True it returns only the eigenvalue, otherwise only returns the eigenvector.

Return numpy.array The largest eigenvector that is the normalized (sum to 1) largest eigenvector as a numpy.array of shape [size] if return_eigenval=False, otherwise returns the eigenvalue as a number.

pyanp.priority.**pri_expeigen**(*mat*, *error=1e-10*)

Calculates priorities using exponential (aka multiplicative) eigenvector

Parameters

- **mat** – An numpy.array of shape [size, size] of pairwise comparisions.
- **error=1e-10** – The convergence error term

Return numpy.array The resulting exponential eigenvector as a numpy.array of shape [size]

pyanp.priority.**pri_geomavg**(*mat*)

Calculates the priorities using the geometric mean method, aka Log Least Squares Method (LLSM).

Parameters **mat** – An numpy.array of dimension [size,size]

Return numpy.array The resulting llsm priority vector as a numpy.array of shape [size]

pyanp.priority.**pri_llsm**(*mat*)

Calculates the priorities using the geometric mean method, aka Log Least Squares Method (LLSM).

Parameters **mat** – An numpy.array of dimension [size,size]

Return numpy.array The resulting llsm priority vector as a numpy.array of shape [size]

pyanp.priority.**ratio_greater_1**(*a*, *b*)
The ratio of a to b (or b to a) that is larger than or equal to 1.

Parameters

- **a** – A numerical value for the ratio calculation
- **b** – Another numerical value

Returns 1 if a or b is 0, otherwise max(a/b, b/a)

pyanp.priority.**ratio_mat**(*pv*) → numpy.ndarray
Returns the ratio matrix of a vector

Parameters **pv** – An array-like object with len(pv)=size

Returns A numpy.array of shape [size, size] of the ratios

pyanp.priority.**size_array_like**(*mat_or_size*)
Returns the size of an array like or integer

Parameters **mat_or_size** – Either an integer (specifying the size) or an array-like or numpy.array of shape [size, size]. If array-like list of lists, we only use len(mat_or_size), we do not check that the array-like is actually square.

Returns The parameter if it was an integer, len(mat_or_size) if param is a list, or mat_or_size.shape[0] if mat_or_size is a numpy.ndarray

pyanp.priority.**utmrowlist_to_npmatrix**(*list_of_votes*) → numpy.ndarray
Convert a list of values to a pairwise matrix, assuming the list is the upper triangular part only

Parameters **list_of_votes** – An array like, the first elements are the top row of the UTM part of the matrix. Then it goes to the second row, etc.

Returns A numpy.array of the full pairwise comparison matrix.

1.4 Limit matrix calculations

All ANP limit matrix functions live here.

Contains all limit matrix calculations

pyanp.limitmatrix.**_mat_pow2**(*mat*, *power*)
Calculates mat^N where $N \geq power$ and N is a power of 2. It does this by squaring mat, and squaring that, etc, until it reaches the desired level. It takes at most floor(log_2(power))+1 matrix multiplications to do this, which is much preferred for large powers.

Parameters

- **mat** – The numpy array to raise to a power.
- **power** – The power to be greater than or equal to

Returns The resulting power of the matrix

pyanp.limitmatrix.**calculus**(*mat*, *error=1e-10*, *max_iters=5000*, *use_hierarchy_formula=True*, *col_scale_type=None*)
Calculates the 'Calculus Type' limit matrix from superdecisions

Parameters

- **mat** – The scaled supermatrix to calculate the limit matrix of
- **error** – The maximum error to allow between iterations
- **max_iters** – The maximum number of iterations before we give up, after we calculate the start power
- **use_hierarchy_formula** – If True and the matrix is for a hierarchy we use that formula instead.
- **col_scale_type** – A string if 'all' it scales mat1 by max(mat1) and similarly for mat2 otherwise, it scales by column

Returns The calculats limit matrix as a numpy array.

pyanp.limitmatrix.**hiearhcy_formula**(*mat*)
Uses the hierarchy formula to calculate the limit matrix. This is essentially the normalization of the sum of higher powers of mat.

Parameters **mat** – A square nump.array that you wish to find the limit matrix of, using the hiearchy formula.

Returns The limit matrix, unless the matrix was not a hiearchy. If the matrix was not a hierarchy we return None

pyanp.limitmatrix.**hierarchy_nodes**(*mat*)
Returns the indices of the nodes that are hierarchy ones. The others are not hierachy

Parameters **mat** – A supermatrix (scaled or non-scaled, both work).

Returns List of indices that are the nodes which are hierarhical.

pyanp.limitmatrix.**limit_newhierarchy**(*mat*, *with_limit=False*, *error=1e-10*, *col_scale_type=None*, *max_count=1000*)
Performs the new hiearchy limit matrix calculation

Parameters

- **mat** – The matrix to perform the calculation on.
- **with_limit** – Do we include the final limit step?

Returns The resulting numpy array

pyanp.limitmatrix.**limit_sinks**(*mat*, *straight_normalizer=True*)
Performs the limit with sinks calculation. We break up the matrix into sinks and nonsinks, and use those pieces.

Parameters

- **mat** – The matrix to do the limit sinks calculation on.
- **straight_normalizer** – If False we normalize at each step, if True we normalize at the end.

Returns The resulting numpy array result.

pyanp.limitmatrix.**normalize**(*mat*, *inplace=False*)

Makes the columns of a matrix add to 1 (unless the column summed to zero, in which case it is left unchanged) Does this by dividing each column by the sum of that column.

Parameters

- **mat** – The matrix to normalize
- **inplace** – If true normalizes the matrix sent in, otherwise it leaves that matrix alone, and returns a normalized copy

Returns If inplace=False, it returns the normalized matrix, leaving the param mat unchanged. Otherwise it returns nothing and normalizes the param mat.

pyanp.limitmatrix.**normalize_cols_dist**(*mat1*, *mat2*, *tmp1=None*, *tmp2=None*, *tmp3=None*, *col_scale_type=None*)

Calculates the distance between matrices mat1 and mat2 after they have been column normalized. tmp1, tmp2, tmp3 are temporary matrices to store the normalized versions of things. This code could be called many times in a limit matrix calculation, and allocating and freeing those temporary storage bits could take a lot of cycles. This way you allocate them once at the top level loop of the limit matrix calculation and they are reused again and again.

If you do not wish to avail yourself of this savings, simply leave them as None's and the algorithm will allocate as appropriate

Parameters

- **mat1** – First matrix to compare
- **mat2** – The other matrix to compare
- **tmp1** – A temporary storage matrix of same size as mat1 and mat2. If None, it will be allocated inside the fx.
- **tmp2** – A temporary storage matrix of same size as mat1 and mat2. If None, it will be allocated inside the fx.
- **tmp3** – A temporary storage matrix of same size as mat1 and mat2. If None, it will be allocated inside the fx.
- **col_scale_type** – A string if 'all' it scales mat1 by max(mat1) and similarly for mat2 otherwise, it scales by column

Returns The maximum difference between the column normalized versions of mat1 and mat2

pyanp.limitmatrix.**priority**(*matrix*, *limit_calc=<function calculus>*)

Calculates the limit matrix and extracts priority from it. Really just a convenience function.

Parameters

- **matrix** – The scaled supermatrix to calculate the priority for
- **limit_calc** – The limit matrix calculation to use

Returns The priority as a series

pyanp.limitmatrix.**priority_from_limit**(*limit_matrix*)

Calculates the priority from a limit matrix, i.e. sums columns and divides by the number of columns.

Parameters **limit_matrix** – The matrix to extract the priority from

Returns 1d numpy array of the priority

pyanp.limitmatrix.**two_two_breakdown**(*mat*, *upper_right_indices*)

Given the indices for the upper right portion of a matrix, break the matrix down into a 2x2 matrix with the submatrices in each piece. Useful for limit matrix calculations that split the problem up into the hierarchical and network components and do separate calculations and then bring them together. :param mat: The matrix to split into

A	B
C	D

form, where A is the "upper_right_indcies" and D is the opposite

Parameters **upper_right_indices** – List of indices of the upper right positions.

Returns A list of [A, B, C, D] of those matrices

pyanp.limitmatrix.**zero_cols**(*full_mat*, *non_zero=False*)

Returns the list of indices of columns that are zero or non_zero depending on the parameter non_zero

Parameters

- **mat** – The matrix to search over
- **non_zero** – If False, returns the indices of columns that are zero, otherwise returns indices of columns that a not zero.

Returns A list of indices of columns of the type determined by the parameter non_zero.

1.5 Example matrices module

This is where we store standard examples of pairwise and supermatrices.

pyanp.exmats.**matrix_matching**(*df=None*, *description=None*, *keywords=None*, *size=None*, *author=None*)

Finds matrices that match search criteria

Parameters

- **df** – The dataframe to search through, either SUPERMATRIX_EXS or PAIRWISE_EXS if None we use SUPERMATRIX_EXS
- **description** – A substring to search through description
- **keywords** – A list of keywords to find
- **size** – The size
- **author** – The contributing author, a substring search

Returns List of indices of the matches in the given dataframe

pyanp.exmats.**pairwisematrix_ex**(*name=None*, *description=None*, *keywords=None*, *size=None*, *author=None*)

Find the pairwise matrix example that matches the conditions

Parameters

- **name** – If not None, we find the single matrix with this name/id
- **description** – Substring search in description

- **keywords** – exact match keywords
- **size** – exact match size
- **author** – substring search author

Returns A single numpy.array item if only one matches the constraint, or a panda.Series indexed by name or the resulting numpy.array s.

pyanp.exmats.**supermatrix_ex**(*name=None*, *description=None*, *keywords=None*, *size=None*, *author=None*)

Find the supermatrix example that matches the conditions

Parameters

- **name** – If not None, we find the single matrix with this name/id
- **description** – Substring search in description
- **keywords** – exact match keywords
- **size** – exact match size
- **author** – substring search author

Returns A single numpy.array item if only one matches the constraint, or a panda.Series indexed by name or the resulting numpy.array s.

1.6 General functions module

Generally useful code goes in this module.

Generally useful math and other functions.

pyanp.general.**get_matrix**(*fname_or_df*, *sheet=0*) → numpy.ndarray

Returns a dataframe from a csv/excel filename (or simply returns the dataframe if it is passed as input

Parameters **fname_or_df** – The file name to get as a dataframe, or a dataframe

(in which case that param is returned)

Parameters **sheet** – If it is a filename, which sheet to use

Returns The dataframe

pyanp.general.**islist**(*val*)

Simple function to check if a value is list like object

Parameters **val** – The object to check its listiness.

Returns Boolean True/False

pyanp.general.**linear_interpolate**(*xs*, *ys*, *x*)

Piecewise linear interpolation between a bunch of x and y coordinates.

Parameters

- **xs** – Increasing x values (no dupes)
- **ys** – The y values
- **x** – The x value to linearly interpolate at

Returns if x < xs[0], returns ys[0], if x > xs[-1], returns ys[-1] else linearly interpolates.

pyanp.general.**matrix_as_df**(*matrix: numpy.ndarray*, *index*) → pandas.core.frame.DataFrame
Returns a square numpy.ndarray as a dataframe with given row/col names

Parameters

- **matrix** – The square numpy array
- **index** – The names of the rows (which is the same as the name of the columns as a list of strings).

Returns The dataframe

pyanp.general.**unwrap_list**(*list_ish*)
Takes something that is a list(list(tuple(e1, e2))) and unwraps til we have (e1, e2).

Parameters **list_ish** – The list to unwrap

Returns Unwrapped version of list

1.7 Direct data class

class pyanp.direct.**Direct**(*alt_names=None*)
Represents the concept of directly setting data. It is a single user only *pyanp.prioritizer.Prioritizer* instance.

__init__(*alt_names=None*)
Initialize self. See help(type(self)) for accurate signature.

add_alt(*alt_name: str*) → None
Adds an alternative.

Parameters **alt_name** – The name of the alt to add.

Returns Nothing

Raises **ValueError** – If the alternative already existed

add_user(*uname: str*) → None
Does nothing since Direct current does not have users.

Parameters **uname** – The name of the user we should add

Returns Nothing

priority(*username=None*, *ptype: pyanp.prioritizer.PriorityType = None*)
Gets the priority for the given user. At the moment it simply ignores user since direct data only stores one data set for all users.

Parameters

- **username** – The name of the user, but it is ignored.
- **ptype** – Should we normalize, idealize, or leave the priority alone.

Returns A pandas.Series whose index is the alternative names and whose values are the priorities.

usernames()
Direct has no notion of users at the moment, so this returns the empty list.

Returns Empty list

1.8 Prioritizer class

class pyanp.prioritizer.**Prioritizer**

This class is the abstract representation of anything that prioritizes a list of items. Examples include *pyanp.pairwise.Pairwise* for doing group pairwise comparisons and *pyanp.ahptree.AHPTree* for doing group AHP tree models.

add_alt(*alt_name: str*, *ignore_existing=True*) → None

Add an alternative to the prioritizer. This should be overriden by the implementing class.

Parameters **alt_name** – The name of the alternative to add.

Returns Nothing

add_user(*uname*)

Adds a user to this prioritizer object.

Parameters **user_name** – The name of the user to add

Returns Nothing

Raises **ValueError** – If the user already existed.

alt_names()

Returns A list of the alts in this prioritizer object.

data_names(*append_to=None*, *post_pend=''*) → str

Return string of newline separated names for the data that this prioritizer needs for each user.

Parameters

- **append_to** – If not none, elements are appended here, otherwise a new list is created.
- **post_pend** – A string to post_pend to each name

Returns List of strings of names.

nalts()

Returns The number of alternatives (things you are pairwise comparing) in this group pairwise comparison object.

priority(*username=None*, *ptype: pyanp.prioritizer.PriorityType = None*) → pandas.core.series.Series

Calculates the alternative priorities. Should be overriden by the implementing class.

Parameters **user_name** – The name/names of the users to calculate the priority

of. If None, we get the priority of the group average. If it is a string, we get the priority of that user. If it is a list of users, we get the priority of the group average for that list of users.

Parameters **ptype** – How should we normalize the resulting priorities (if at all).

Returns A pandas.Series whose indices are the alternative names and whose values are the priorities of those alternatives.

priority_df(*user_infos=None*) → pandas.core.frame.DataFrame

Returns the priority scores dataframe for all users and the group

Parameters **user_infos** – A list of users to do this for, if None is a part of this list, it means group average. If None, it defaults to None plus all users.

Returns pandas.DataFrame rows are alternatives, cols are users.

user_names()

Returns A list of the users in this prioritizer object.

class pyanp.prioritizer.**PriorityType**
An enumeration telling how to normalize priorities for a calculation

IDEALIZE = 3
Divide priorities by max, so that the largest is 1.

NORMALIZE = 2
Divide priorities by sum, so that they sum to 1.

RAW = 1
Leave the priorities unchanged.

apply(*vals*)
Returns a copy of the parameter vals that has been adjusted as this PriorityType would.

Parameters **vals** – A list-like object of values. We return a copy that is adjusted.

Returns A list-like of the same type as 'vals' that has been normalized as

this PriorityType would do.

1.9 ANP Row Sensitivity

All ANP row sensitivity calculations are in this module.

pyanp.rowsens.**calcp0**(*mat*, *row*, *cluster_nodes*, *orig*, *p0mode*)
Calculates the p0, or resting, value for the row sensitivity

Parameters

- **mat** – The matrix to do row sensitivity on
- **row** – The row to do row sensitivity on
- **cluster_nodes** – The indices of the other nodes in the cluster that *row* is in. Used for *influence_marginal()* if p0mode is an integer meaning smart mode for alt=p0mode :param orig: The original weight, used if p0mode is not an integer (meaning smart) or a float (meaning a direct p0 value).
- **p0mode** – This controls the calculation and has 3 cases: Case 1: if it is a float, you are directly setting the p0 value to whatever p0mode is. Case 2: if it is an integer, this is the smart p0 mode, and it treats p0mode as the index of the alternative/node to make continuous. Case 3: otherwise we assume you want original weights to be the p0 value, and return the parameter *orig*

Returns The p0 or resting value, see the *p0mode* parameter for more information

pyanp.rowsens.**influence_fixed**(*mat*, *row*, *cluster_nodes=None*, *influence_nodes=None*, *delta=0.25*, *p0mode=0.5*, *limit_matrix_calc=<function calculus>*)
Calculates fixed influence, i.e. we do row sensitivity and calculate the difference

Parameters

- **mat** – The scaled supermatrix to perform the calculation on
- **row** – The row to use for anp row sensitivity
- **cluster_nodes** – If you wish to normalize by cluster, this should be the indices of the nodes that are in row's cluster (including row itself).

- **influence_nodes** – The indices of the nodes to calculate the influence of, with respect to row. If None it calculates the influence of all nodes other than row.
- **delta** – How much to change from p0 for the fixed influence
- **p0mode** – This controls the calculation and has 3 cases: Case 1: if it is a float, you are directly setting the p0 value to whatever p0mode is. Case 2: if it is an integer, this is the smart p0 mode, and it treats p0mode as the index of the alternative/node to make continuous. Case 3: otherwise we assume you want original weights to be the p0 value, and return the parameter *orig*
- **limit_matrix_calc** –

Returns A pandas.Series whose index is influence_nodes and whose values are the influence scores of those nodes with respect to the row.

pyanp.rowsens.**influence_limit**(*mat*, *row*, *cluster_nodes=None*, *influence_nodes=None*, *delta=1e-06*, *p0mode=0.5*, *limit_matrix_calc=<function calculus>*)

Calculates the limit influence score of the influence_nodes with respect to row.

Parameters

- **mat** – The scaled supermatrix to perform the calculation on
- **row** – The row to use for anp row sensitivity
- **cluster_nodes** – If you wish to normalize by cluster, this should be the indices of the nodes that are in row's cluster (including row itself).
- **influence_nodes** – The indices of the nodes to calculate the influence of, with respect to row. If None it calculates the influence of all nodes other than row.
- **delta** – We use 1-delta for the p-value to plugin to approximate the limit as p -> 1
- **p0mode** – This controls the calculation and has 3 cases: Case 1: if it is a float, you are directly setting the p0 value to whatever p0mode is. Case 2: if it is an integer, this is the smart p0 mode, and it treats p0mode as the index of the alternative/node to make continuous. Case 3: otherwise we assume you want original weights to be the p0 value, and return the parameter *orig*
- **limit_matrix_calc** – A function which takes a single input, the matrix to take the limit of.

Returns A tuple of 2 items, the first is a pandas.Series whose indices are 'Node 1', 'Node 2' (and the indices after "Node " are the influence_node indices) and whose values are the limit value. The second element of the returned tuple is a pandas.Series with the same indices and whose values are the p0 values we used for that alternative.

pyanp.rowsens.**influence_marginal**(*mat*, *row*, *influence_nodes=None*, *cluster_nodes=None*, *left_or_right=None*, *delta=1e-06*, *p0mode=0.5*, *limit_matrix_calc=<function calculus>*)

Calculates the marginal influence

Parameters

- **mat** – The scaled supermatrix to calculate marginal influence on
- **row** – The index of the row to perform the marginal influence on
- **influence_nodes** – The nodes to calculate the marginal influence of the row upon, if None then it assumes all nodes except row.
- **cluster_nodes** – The other nodes in the parameter row's cluster (including row itself), so we can scale by cluster. If None we do not scale by cluster.

- **left_or_right** – An integer telling whether we should do left-hand side derivative, right-hand side derivative or average them. If left_or_right < 0, then we do LHS deriv. If left_or_right > 0, we do RHS deriv. Finally, if left_or_right == 0, we average LHS and RHS.
- **delta** – The delta_x to use for the derivative calculation.
- **p0mode** – This controls the calculation and has 3 cases: Case 1: if it is a float, you are directly setting the p0 value to whatever p0mode is. Case 2: if it is an integer, this is the smart p0 mode, and it treats p0mode as the index of the alternative/node to make continuous. Case 3: otherwise we assume you want original weights to be the p0 value, and return the parameter *orig*
- **limit_matrix_calc** – A function which takes a single input, the matrix to take the limit of.

Returns A pandas.Series whose indices are influence_nodes and whose values are the marginal influence scores of those nodes with respect to the given row.

pyanp.rowsens.**influence_rank**(*mat*, *row*, *cluster_nodes=None*, *influence_nodes=None*, *limit_matrix_calc=<function calculus>*, *rank_change_nodes=None*, *error=1e-05*, *upper_lower_both=0*, *round_to_decimal=5*, *return_full_info=False*)

Calculates rank influence scores.

Parameters

- **mat** – The scaled supermatrix to perform the calculation on
- **row** – The row to use for anp row sensitivity
- **cluster_nodes** – If you wish to normalize by cluster, this should be the indices of the nodes that are in row's cluster (including row itself).
- **influence_nodes** – The indices of the nodes to calculate the influence of, with respect to row. If None it calculates the influence of all nodes other than row.
- **limit_matrix_calc** – A function with one parameter, that calculates the limit matrix.
- **rank_change_nodes** – The nodes to look for rank change at
- **error** – While we narrow down our search for the p-value that causes a rank change, how close do we want the values between a change happening and not, to be.
- **upper_lower_both** – Do we want to: Case 1: look for rank change only by changing p > 0.5=p0, if so upper_lower_both > 0. Case 2: look for rank change only by changing p < 0.5=p0, if so upper_lower_both < 0. Case 3: look for rank change by changing p>0.5 and p<0.5, if so upper_lower_both = 0.
- **round_to_decimal** – How many decimals should we round the score to for ranking purposes
- **return_full_info** – If True returns more info, see the return section for more details

Returns

A list of one or more numbers, controlled by return_full_info and upper_lower_both:

- If upper_lower_both < 0: our p-val search will only be for pval < 0.5
  - If return_full_info is True: We return pval_where_rank_chg_happens, score_of_that_pval
  - Otherwise: We return score_of_that_pval
- If upper_lower_both > 0: our pval search will only be for pval > 0.5

  - If return_full_info is True: We return pval_where_rank_chg_happens, score_of_that_pval
  - Otherwise: We return score_of_that_pval
- If upper_lower_both = 0: we check both lower and upper
  - If return_full_info is True: We return max_of_upper_lower_scores, lower_rank_value, lower_rank_chg_score, upper_rank_value, upper_rank_chg_score
  - Otherwise we return: max_of_upper_lower_scores

pyanp.rowsens.**influence_table**(*mat*, *row*, *pvals=None*, *cluster_nodes=None*, *influence_nodes=None*, *p0mode=None*, *limit_matrix_calc=<function calculus>*, *graph=True*, *return_p0vals=False*, *node_names=None*)

Calculates the direct influence score, i.e. it calculates anp row sensitivity for each of pvals values and stores the new scores of the influence_nodes.

Parameters

- **mat** – The scaled supermatrix to perform the calculation on
- **row** – The row to use for anp row sensitivity
- **pvals** – The values to set p to, this should be a list (or list like) object of values before 0 and 1.
- **cluster_nodes** – If you wish to normalize by cluster, this should be the indices of the nodes that are in row's cluster (including row itself).
- **influence_nodes** – The indices of the nodes to calculate the influence of, with respect to row. If None it calculates the influence of all nodes other than row.
- **p0mode** – This controls the calculation and has 3 cases: Case 1: if it is a float, you are directly setting the p0 value to whatever p0mode is. Case 2: if it is an integer, this is the smart p0 mode, and it treats p0mode as the index of the alternative/node to make continuous. Case 3: otherwise we assume you want original weights to be the p0 value, and return the parameter *orig*
- **limit_matrix_calc** – A function which takes a single input, the matrix to take the limit of.
- **graph** – If True, we return a matplotlib graph, otherwise we return pandas.DataFrame, p0vals
- **return_p0vals** – If true and not doing graphing, we return a tuple of the dataframe of the results, and the 2nd item as Series whose index is the names of the nodes, and whose values are the (x,y) position of the resting p0 value
- **node_names** – If None, we use Node 0, Node 1, ... to label nodes, otherwise we use this.

Returns

If graph=True, we return nothing, but create a matplotlib object and call plt.show(). Otherwise if return_p0vals is True we return a pair of items. The first is the dataframe of results, whose indices are "Node 1", "Node 2", ... which corresponds to influence_nodes (and the indices after "Node " are the influence_node indices) and has 2 columns, 'x' is the pvals and 'y' is the resulting influence score (i.e. changed priority). The second element is a pd.Series whose indices is the same as the dataframe and whose values are pairs of items (x,y) where x is the p0 value for the given alternative and the y is the influence score of that alternative at that p-value.

If return_p0vals is False we return the first dataframe item only.

pyanp.rowsens.**influence_table_plot**(*df*, *p0s*)
Graphs the return value of influence_table(graph=False), useful if you want to have both the graph done and also the table of values. :param df: The 1st returned component from influence_table(graph=False): a dataframe :param p0s: The 2nd returned component from influence_table(graph=False): a Series of (x,y)'s :return: Nothing, but does call plt.show() to make the matplotlib graph visible.

pyanp.rowsens.**p0mode_is_direct**(*p0mode_value*)
Is the p0mode value a "directly set value". See calcp0 for more info on p0mode values.

Parameters **p0mode_value** – If it is an float, this p0mode value is the direct value to use for p0.

Returns True | False

pyanp.rowsens.**p0mode_is_smart**(*p0mode_value*) → bool
Is the p0mode value a "smart value". See calcp0 for more info on p0mode values.

Parameters **p0mode_value** – If it is an int, this p0mode value represents doing smart p0 making the node of that index (i.e.p0mode's value) smooth

Returns True | False

pyanp.rowsens.**p0mode_name**(*p0mode_value*) → str
Tells what kind of p0 the p0mode value is

Parameters **p0mode_value** – The p0mode value to get a name of

Returns A string/human readable bit of information about the p0mode.

pyanp.rowsens.**rank_change**(*vec1*, *vec2*, *places_to_rank*, *rank_change_places=None*, *round_to_decimal=5*)
A calculation that rounds 2 vectors to round_to_decimal places and then looks to see if the ranking of rounded vec1 is different from rounded vec2.

Parameters

- **vec1** – A list or list-like object to check rank changing
- **vec2** – Another list or list-like object to check rank changing
- **places_to_rank** – Indices to rank.
- **rank_change_places** – Of the indices we are ranking, which ones are we checking for a change (if None we check all indices for rank change).
- **round_to_decimal** – The number of decimal places to round to, before checking for rank changes

Returns True if a rank change happen and False otherwise

pyanp.rowsens.**row_adjust**(*mat*, *row*, *p*, *cluster_nodes=None*, *inplace=False*, *p0mode=None*)
Performs an actual row adjust on the matrix, either inplace or returns an adjusted copy, leaving the original unchanged.

Parameters

- **mat** – The scaled supermatrix to perform anp row sensitivity on.
- **row** – The row index to perform the anp row sensitivity on
- **p** – The p value to adjust to
- **cluster_nodes** – The other nodes in the parameter row's cluster (including row itself), so we can scale by cluster. If None we do not scale by cluster.
- **inplace** – Should we change the matrix mat, or should we create a new one, adjust it, and return it?

- **p0mode** – See calcp0() function

Returns The adjusted matrix if inplace=False, and otherwise returns nothing and changes the matrix mat.

pyanp.rowsens.**row_adjust_priority**(*mat*, *row*, *p*, *cluster_nodes=None*, *p0mode=None*, *limit_matrix_calc=<function calculus>*, *normalize_to_orig=True*)

Adjusts a row of matrix and recalculates the priorities of all the nodes.

Parameters

- **mat** – The scaled supermatrix to perform the calculation on
- **row** – The row to use for anp row sensitivity
- **cluster_nodes** – If you wish to normalize by cluster, this should be the indices of the nodes that are in row's cluster (including row itself).
- **p0mode** – This controls the calculation and has 3 cases: Case 1: if it is a float, you are directly setting the p0 value to whatever p0mode is. Case 2: if it is an integer, this is the smart p0 mode, and it treats p0mode as the index of the alternative/node to make continuous. Case 3: otherwise we assume you want original weights to be the p0 value, and return the parameter *orig*
- **limit_matrix_calc** – A function which takes a single input, the matrix to take the limit of.
- **normalize_to_orig** – If True we normalize the returning priority score so that the [row] index of it has the same value as the original and the other values are rescaled. Otherwise we simply normalize the priority vector directly.

1.10 Rating class

Class for all rating related things.

class pyanp.rating.**Rating**

Represents rating a full group of alternatives for a group of users. The data is essentially a dataframe and a WordEval object to evaluate that to scores.

add_alt(*alt_name*, *ignore_existing=True*)

Adds an alternative/s, by name

Parameters

- **alt_name** – A str name, or a list of names to add.
- **ignore_existing** – If True and we try to add an existing alternative we simply skip by, otherwise we throw an error.

Returns Nothing

add_user(*uname*)

Adds one or more uses to this system.

Parameters **uname** – The str name of the user to add, or a list of str names of users to add.

Returns Nothing

alt_names()

Returns A list of str alternative names in this system. Ordered as the data in the ratings votes are ordered (columns).

is_alt(*alt: str*) → bool
Tells if the item is an alternative

Parameters **alt** – The name of the alternative to check for.

Returns True/False

is_user(*uname: str*)

Parameters **uname** – The name of the user to check for

Returns True/False if the given user exists in the system.

nalts() → int

Returns The number of alternatives in this system.

nusers() → int
The number of users in this system.

Returns The number of users

priority(*username=None*, *ptype: pyanp.prioritizer.PriorityType = None*)
Calculates the alternative priority for the specified user/users and the given normalizer type.

Parameters

- **username** – The name (this of names) of the user (users) to get the overall priority of. If None, then we return the total group average.
- **ptype** – How should we normalize?

Returns A pandas.Series whose index is self.alt_names() and whose values are the priorities.

set_word_eval(*param*)
Sets the WordEval object

Parameters **param** – This could either be a WordEval object, or a something that WordEval(param) would work with

Returns None

user_names()

Returns A list of str names of users in this system. Ordered as the data in the ratings votes are ordered (the rows).

vote_column(*alt_name*, *votes*, *createUnknownUsers=True*)
Specifies all votes (across all users) for a specific alternative.

Parameters

- **alt_name** – The name of the alternative to set the data for
- **votes** – Should either be a list with self.nusers() items, or a pandas.Series or dict with usernames as index.
- **createUnknownUsers** – If True and unknown users appear in the index of votes, we will create those users before trying to do the assignment.

Returns Nothing

vote_values(*username=None*, *alt_name=None*)
Gets the numeric vote values for the given user/alternative (or whole column or dataframe).

Parameters

- **username** – If None, we get the values for all users. If a list get the values for each user in the list, or it could just be a single username.
- **alt_name** – Either None, meaing get it for all alternatives, or a single alternative name (to get one column).

Returns If username=None and alt_name=None, returns a pandas.DataFrame of the numeric values. Otherwise returns a pandas.Series of values as the result.

class pyanp.rating.**WordEval**(*vals*)

Information for a Word Evaluator, i.e. a function that inputs a word and outputs a numeric value.

eval(*word*)

Evaluates a word, or a pandas.Series of words.

Parameters **word** – The string word to evaluate to a number, or a pandas.Series of data.

Returns The float value if we can evaluate, or None if a single value is passed in. If the word was actually a pandas.Series, we return a pandas.Series with the same index.

get_key(*word*)

Find the key word for this word. A WordEval has a list of words that represent different levels/numerical values. Those words are called keys. In addition, each key has a list of synonyms. For instance the keyword "high" might have a synonym "hi" or "h". In that case get_key("hi") would return "high".

Parameters **word** – The word to look up a synonym for.

Returns The key if this word is a key or a synonym. If it is not a synonym or key, we return None.

keys_match_score(*word_list*)

This function tells us how well this WordEval interprets a list of words. It is used for searhcing through the "standard list" of words to find the best match for a data set.

Parameters **word_list** – The list-like of words to see how we can match.

Returns A score <= 1. A positive number means no missing words, i.e. every word in word_list has a value in this WordEval object. The larger number means our word_list uses more of the names in this WordEval object.

class pyanp.rating.**WordEvalType**

What kind of WordEval will we use.

pyanp.rating.**best_std_word_evaluator**(*list_of_words*, *return_name=True*)

Finds the WordEval in STD_WORD_EVALUATOR that best matches the list of words

Parameters

- **list_of_words** – The list of words to look for best matches of
- **return_name** – Should we return the best WordEval or its name in the STD_WORD_EVALUATOR.

Returns The name of the best match, or the best match WordEval

pyanp.rating.**clean_word**(*word: str*) → str

Cleans a word before subjecting it to ratings lookup

Parameters **word** – The word to clean.

Returns The sanitized word

1.11 ANP structure module

Group enabled ANPNetwork class and supporting classes.

class pyanp.anp.**ANPCluster**(*network*, *name: str*)
A cluster in an ANP object

Parameters

- **network** – The ANPNetowrk object this cluster is in.
- **name** – The name of the cluster to create.

add_node(**nodes*) → None
Adds one or more nodes

Parameters **nodes** – A vararg list of node names to add to this cluster. The names should all be strings.

Returns Nonthing

cluster_connect(*dest_cluster*) → None
Make a cluster->cluster connection from this node to the destination.

Parameters **dest_cluster** – Either the ANPCluster object to connect to, or the name of the destination cluster.

Returns

data_names(*append_to=None*)
Used when exporting an Excel header for a network, for its data.

Parameters **append_to** – If not None, append header strings to this list. Otherwise we create a new list to append to.

Returns List of strings of comparison name headers. If append_to is not None, we return append_to with the new string headers appended.

is_node(*node_name: str*) → bool
Does a node by that name exist in this cluster

Parameters **node_name** – The name of the node to look for

Returns True/False

nnodes() → int

Returns The number of nodes in this cluster.

node_names() → list

Returns List of the string names of the nodes in this cluster

node_obj(*node_name*)
Get a node in this cluster.

Parameters **node_name** – The node as either a string name, integer position, or simply the ANPObject, in which case there is nothing to do except return it.

Returns ANPNode object. If it wasn't found, None is returned.

node_objs() → list

Returns List of the ANPNode objects in this cluster.

set_prioritizer_type(*prioritizer_class*) → None
Sets the cluster prioritizer type

Parameters **prioritizer_class** – The new type

Returns None

class pyanp.anp.**ANPNetwork**(*create_alts_cluster=True*)
Represents an ANP prioritizer. Has clusters/nodes, comparisons, etc.

Parameters **create_alts_cluster** – If True (which is the default) we start with a cluster that is the alternatives cluster. Otherwise the model starts empty.

add_alt(*alt_name: str*)
Adds an alternative to the model: 1. Adds the altenrative to alts_cluster if not None 2. For each node with a subnetwork, we add the alternative to that subnetwork.

Parameters **alt_name** – The name of the alternative to add

Returns Nothing

add_cluster(**args*) → pyanp.anp.ANPCluster
Adds one or more clusters to a network

Parameters **args** – Can be either a single string, or a list of strings

Returns ANPCluster object or list of ANPCluster objects

add_node(*cl*, **nodes*)
Adds nodes to a cluster

Parameters

- **cl** – The cluster name or object
- **nodes** – The name or names of the nodes

Returns Nothing

add_user(*uname*, *ignore_dupe=False*)
Adds a user to the system

Parameters **uname** – The name of the new user

Returns Nothing

:raise ValueError If the user already existed

alt_names() → list

Returns List of alt names in this ANP model

cluster_incon_std_df(*user_infos=None*) → pandas.core.frame.DataFrame

Parameters **user_infos** – A list of users to do this for, if None is a part of this list, it means group average. If None, it defaults to None plus all users.

Returns DataFrame whose columns are clusters, rows are users (as controlled by user_infos params) and the value is the inconsistency for the given user on the given comparison.

cluster_names() → list

Returns List of string names of the clusters

cluster_obj(*cluster_info: Union[pyanp.anp.ANPCluster, str]*) → pyanp.anp.ANPCluster
Returns the cluster with given information

Parameters **cluster_info** – Either the name of the cluster object to get or the cluster object, or its int position

Returns The ANPCluster object

cluster_objs() → list

Returns List of ANPCluster objects in the network

cluster_prioritizer(*wrtcluster=None*)
Gets the prioritizer for the clusters wrt a given cluster.

Parameters **wrtcluster** – WRT cluster identifier as expected by cluster_obj() function. If None, then we return a dictionary indexed by cluster names and values are the prioritizers

Returns THe prioritizer for that cluster, or a dictionary of all cluster prioritizers

data_names()
Returns the column headers needed to fill in the data for this model

Returns A list of strings that would be usable in excel for parsing headers

global_priority(*username=None*) → pandas.core.series.Series

Parameters **username** – If None, gets it for all users. Otherwise gets it for the user specified. It can also be a list of users, in which case we combine them, as per the theory.

Returns The global priorities Series, index by node name

global_priority_df(*user_infos=None*) → pandas.core.frame.DataFrame

Parameters **user_infos** – A list of users to do this for, if None is a part of this list, it means group average. If None, it defaults to None plus all users.

Returns The global priorities dataframe. Rows are the nodes and columns are the users. The first user/column is the Group Average

has_subnet() → bool

Returns True/False telling if some node had a subentwork

import_pw_series(*series: pandas.core.series.Series*) → None
Takes in a well titled series of data, and pushes it into the right node's prioritizer (or cluster). The name should be A vs B wrt C, where A, B, C are node or cluster names.

Parameters **series** – The series of data for each user. Index is usernames. Values are the votes.

Returns Nothing

import_rating_series(*series: pandas.core.series.Series*)
Takes in a well titled series of data, and pushes it into the right node's prioritizer as ratings (or cluster). Title should be A wrt B, where A and B are either both node names or both column names.

Parameters **series** – The series of data for each user. Index is usernames. Values are the votes.

Returns Nothing

invert_priority(*p*)
Makes a copy of the list like element p, and inverts. The current standard inversion is 1-p. There could be others implemented later.

Parameters **p** – The list like to invert

Returns New list-like of same type as p, with inverted priorities

is_alt(*altname*) → bool
Checks if an alternative exists

Parameters **altname** – The alterantive name to look for

Returns bool

is_user(*uname*) → bool
Checks if a user exists

Parameters **uname** – The name of the user to check for

Returns bool

limit_matrix(*username=None*, *as_df=False*)

Parameters

- **username** – If None, gets it for all users. Otherwise gets it for the user specified. It can also be a list of users, in which case we combine them, as per the theory.
- **as_df** – If True, returns as a dataframe with index and column names as the names of the nodes in the network. Otherwise just returns the array.

Returns The limit supermatrix

nclusters() → int

Returns The number of clusters in the network.

nnodes(*cluster=None*) → int
Returns the number of nodes in the network, or a cluster.

Parameters **cluster** – If None, we return the number of nodes in the network. Otherwise this is the integer position, string name, or ANPCluster object of the cluster to get the node count within.

Returns The count.

node_connect(*src_node*, *dest_node*)
connects 2 nodes

Parameters

- **src_node** – Source node as prescribed by node_object() function
- **dest_node** – Destination node as prescribed by node_object() function

Returns Nothing

node_connection_matrix(*new_mat: numpy.ndarray = None*)
Returns the current node conneciton matrix if new_mat is None. Otherwise, for each item [row, col] in the matrix with a value of 1 we connect from node[row] to node[col].

Parameters **new_mat** – The new node connection matrix. If None, we return the current one.

Returns Current connection matrix.

node_connections() → numpy.ndarray
Returns the node conneciton matrix for this network. :return: A numpy array of shape [nnode, nnodes] where item [row, col]

1 means there is a node connection from col -> row, and 0 means no connection.

node_incon_std_df(*user_infos=None*) → pandas.core.frame.DataFrame

Parameters **user_infos** – A list of users to do this for, if None is a part of this list, it means group average. If None, it defaults to None plus all users.

Returns DataFrame whose columns are (node,cluster) pairs, rows are users (as controlled by user_infos params) and the value is the inconsistency for the given user on the given comparison.

node_invert(*node*, *value=None*)
Either sets, or tells if a node is inverted

Parameters

- **node** – The node to do this on, as expected by node_obj() function
- **value** – If None, we return the boolean about if this node is inverted. Otherwise specifies the new value.

Returns T/F if value=None, telling if the node is inverted. Otherwise returns nothing.

node_names(*cluster=None*) → list
Returns a list of nodes in this network, organized by cluster

Parameters **cluster** – If None, we get all nodes in network, else we get nodes in that cluster, otherwise format as specified by cluster_obj() function.

Returns List of strs of node names

node_obj(*node_name*) → pyanp.anp.ANPNode
Gets the ANPNode object of the node with the given name

Parameters **node_name** – The name of the node to get, or it's overall integer position, or the ANPNode object itself

Returns The ANPNode if it exists, or None

node_objs() → list
Returns a list of ANPNodes in this network, organized by cluster

Returns List of strs of node names

node_objs_with_subnet()

Returns List of ANPNode objects in this network that have v1 subnets

node_prioritizer(*wrtnode=None*, *cluster=None*)
Gets the prioritizer for node->cluster connection

Parameters

- **wrtnode** – The node as understood by node_obj() function.
- **cluster** – Cluster as understood by cluster_obj() function.

Returns If both wrtnode and cluster are specified, a single node prioritizer is returned for that comparison (or None if there was nothing there). Otherwise it returns a dictionary indexed by [wrtnode, cluster] and whose values are the prioritizers for that (only the non-None ones).

nusers() → int

Returns The number of users

priority(*username=None*, *ptype: pyanp.prioritizer.PriorityType = None*) → pandas.core.series.Series
Synthesize and return the alternative scores

Parameters

- **username** – If None, gets it for all users. Otherwise gets it for the user specified. It can also be a list of users, in which case we combine them, as per the theory.
- **ptype** – The priority type to use

Returns A pandas.Series indexed on alt names, values are the score

scaled_supermatrix(*username=None*, *as_df=False*) → numpy.ndarray

Parameters

- **username** – If None, gets it for all users. Otherwise gets it for the user specified. It can also be a list of users, in which case we combine them, as per the theory.
- **as_df** – If True, returns as a dataframe with index and column names as the names of the nodes in the network. Otherwise just returns the array.

Returns The scaled supermatrix

set_alts_cluster(*new_cluster*)
Sets the new alternatives cluster

Parameters **new_cluster** – Cluster specified as cluster_obj() expects.

Returns Nothing

set_pairwise_from_supermatrix(*mat*, *username='Imported'*)
Sets up all pairwise comparisons from supermatrix

Parameters **mat** – As numpy array

Returns Nothing

structure_priority(*username=None*, *ptype: pyanp.prioritizer.PriorityType = None*, *alt_names=None*) → pandas.core.series.Series

subnet(*wrtnode*)
Makes wrtnode have a subnetwork if it did not already.

Parameters **wrtnode** – The node to give a subnetwork to, or get the subnetwork of. Node specified as node_obj() function expects.

Returns The ANPNetwork that is the subnet of this node

subnet_synthesize(*username=None*, *ptype: pyanp.prioritizer.PriorityType = None*)
Does the standard V1 subnetowrk synthesis.

Parameters **username** – The user/users to synthesize for. If None, we group synthesize across all. If a single user, we sythesize for that user across all. If it is a list, we synthesize for the group that is that list of users.

Returns Nothing

synthesize_combine(*priorities: pandas.core.series.Series*, *alt_scores: dict*)
Performs the actual sythesis step from anp v1 synthesis.

Parameters

- **priorities** – Priorities of the subnetworks
- **alt_scores** – Alt scores as dictionary, keys are subnetwork names values are Series whose keys are alt names.

Returns Series whose keys are alt names, and whose values are the synthesized scores.

unscaled_supermatrix(*username=None*, *as_df=False*) → numpy.array

Parameters

- **username** – If None, gets it for all users. Otherwise gets it for the user specified. It can also be a list of users, in which case we combine them, as per the theory.
- **as_df** – If True, returns as a dataframe with index and column names as the names of the nodes in the network. Otherwise just returns the array.

Returns The unscaled supermatrix as a numpy.array of shape [nnode, nnodes]

user_names() → list

Returns List of names of the users

class pyanp.anp.**ANPNode**(*network*, *cluster*, *name: str*)

A node inside a cluster, inside a netowrk. The basic building block of an ANP netowrk.

Parameters

- **network** – An ANPNetwork object that this node lives inside.
- **cluster** – An ANPCluster object that this node lives inside.
- **name** – The name of this node.

data_names(*append_to=None*)

Used when exporting an Excel header for a network, for its data.

Parameters **append_to** – If not None, append header strings to this list. Otherwise we create a new list to append to.

Returns List of strings of comparison name headers. If append_to is not None, we return append_to with the new string headers appended.

get_node_prioritizer(*dest_node*, *create=False*, *create_class=<class 'pyanp.pairwise.Pairwise'>*, *dest_is_cluster=False*) → pyanp.prioritizer.Prioritizer

Gets the node prioritizer for the other_node

Parameters **dest_node** – The node as a int, str, or ANPNode object.

Returns The prioritizer if it exists, or None

get_unscaled_column(*username=None*) → pandas.core.series.Series

Returns the column in the unscaled supermatrix for this node.

Parameters **username** – The user/users to do this for. Typical Prioritizer calculation usage, i.e. None means do for all group average.

Returns A pandas series indexed by the node names.

is_node_cluster_connection(*dest_cluster: str*) → bool

Is this node connected to a cluster.

Parameters **dest_cluster** – The name of the cluster

Returns True/False

is_node_node_connection(*dest_node*) → bool

Checks if there is a node connection from this node to dest_node

Parameters **dest_node** – The node as a int, str, or ANPNode object.

Returns

node_connect(*dest_node*) → None

‘ Make a node connection from this node to dest_node

Parameters **dest_node** – The destination node as a str, int, or ANPNode. It can be a list of nodes, and then we will coonect each node from this node. The dest_node should be in any format accepted by ANPNetwork._get_node()

set_node_prioritizer_type(*destNode*, *prioritizer_class*)
Sets the node prioritizer type

Parameters

- **destNode** – An ANPNode object, string, or integer location
- **prioritizer_class** – The new type

Returns None

pyanp.anp.**anp_from_dict**(*cluster_dict: dict*) → pyanp.anp.ANPNetwork
Creates an ANPNetwork from a dictionary whose keys are cluster names and whose values are list of node names in that cluster

Parameters **cluster_dict** – Keys are cluster names. If the cluster name starts with *, that is the alternatives cluster (and the asterisk is removed from the name). The values are list of strings that are the names of the nodes in that network

Returns The ANPNetwork with that structure

pyanp.anp.**anp_from_excel**(*excel_fname: str*) → pyanp.anp.ANPNetwork
Parses an excel file to get an ANPNetwork

Parameters **excel_fname** – The name of the excel file

Returns The newly created ANPNetwork object

pyanp.anp.**anp_from_scaled_supermatrix**(*supermatrix*)
Creates an ANPNetwork object from a scaled supermatrix. We do this by: 1. Parsing the supermatrix and using row sums across columns to figure out clusters 2. We use pairwise comparison objects for each node and cluster comparison 3. We use the ratio of the priorities from the supermatrix to get the votes

Parameters **supermatrix** – The super matrix

Returns

pyanp.anp.**anp_manual_scales_from_excel**(*anp: pyanp.anp.ANPNetwork*, *excel_fname*)
Parses manual rating scales from an Excel file

Parameters

- **anp** – The model to put the scale values in.
- **excel_fname** – The string file name of the excel file with the data

Returns Nothing

pyanp.anp.**clean_name**(*name: str*) → str

Cleans up a string for usage by:

1. stripping off begging and ending spaces
2. All spaces convert to one space
3. and

are treated like a space

param name The string name to be cleaned

return The cleaned name.

pyanp.anp.**get_item**(*tbl: dict*, *key*)
Looks up an item in a dictionary by key first, assuming the key is in the dictionary. Otherwise, it checks if the key is an integer, and returns the item in that position.

Parameters

- **tbl** – The dictionary to look in
- **key** – The key, or integer position to get the item of

Returns The item, or it not found, None

pyanp.anp.**is_pw_col_name**(*col: str*) → bool
Checks to see if the name matches the naming convention for a pairwise comparison, i.e. A vs B wrt C

Parameters **col** – The title of the column to check

Returns T/F

pyanp.anp.**is_rating_col_name**(*col: str*) → bool
Checks to see if the name matches the naming convention for a rating column of data, i.e. A wrt B

Parameters **col** – The name of the column

Returns T/F

pyanp.anp.**sum_subnetwork_formula**(*priorities: pandas.core.series.Series*, *dict_of_series: dict*)
A function that takes the weighted sum of values. Used for synthesis.

Parameters

- **priorities** – Series whose index are the nodes with subnetworks and values are their weights.
- **dict_of_series** – A dictionary whose keys are the same as the keys of priorities, i.e. the nodes with subnetworks. The values are Series whose keys are alternative names and values are the synthesized alternative scores under that subnetwork.

Returns

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

pyanp.anp, 26
pyanp.exmats, 14
pyanp.general, 15
pyanp.limitmatrix, 11
pyanp.pairwise, 5
pyanp.priority, 8
pyanp.rating, 23
pyanp.rowsens, 18

Index

Symbols

__init__() (*pyanp.ahptree.AHPTree method*), 1
__init__() (*pyanp.direct.Direct method*), 16
_mat_pow2() (*in module pyanp.limitmatrix*), 11

A

add_alt() (*pyanp.ahptree.AHPTree method*), 1
add_alt() (*pyanp.anp.ANPNetwork method*), 27
add_alt() (*pyanp.direct.Direct method*), 16
add_alt() (*pyanp.pairwise.Pairwise method*), 5
add_alt() (*pyanp.prioritizer.Prioritizer method*), 17
add_alt() (*pyanp.rating.Rating method*), 23
add_child() (*pyanp.ahptree.AHPTree method*), 1
add_cluster() (*pyanp.anp.ANPNetwork method*), 27
add_node() (*pyanp.anp.ANPCluster method*), 26
add_node() (*pyanp.anp.ANPNetwork method*), 27
add_place() (*in module pyanp.pairwise*), 8
add_user() (*pyanp.ahptree.AHPTree method*), 1
add_user() (*pyanp.anp.ANPNetwork method*), 27
add_user() (*pyanp.direct.Direct method*), 16
add_user() (*pyanp.pairwise.Pairwise method*), 5
add_user() (*pyanp.prioritizer.Prioritizer method*), 17
add_user() (*pyanp.rating.Rating method*), 23
AHPTree (*class in pyanp.ahptree*), 1
alt_direct() (*pyanp.ahptree.AHPTree method*), 1
alt_incon_std() (*pyanp.ahptree.AHPTree method*), 2
alt_index() (*pyanp.pairwise.Pairwise method*), 6
alt_names() (*pyanp.anp.ANPNetwork method*), 27
alt_names() (*pyanp.pairwise.Pairwise method*), 6
alt_names() (*pyanp.prioritizer.Prioritizer method*), 17
alt_names() (*pyanp.rating.Rating method*), 23
alt_pwmatrix() (*pyanp.ahptree.AHPTree method*), 2
altpw() (*pyanp.ahptree.AHPTree method*), 2
anp_from_dict() (*in module pyanp.anp*), 33
anp_from_excel() (*in module pyanp.anp*), 33
anp_from_scaled_supermatrix() (*in module pyanp.anp*), 33
anp_manual_scales_from_excel() (*in module pyanp.anp*), 33
ANPCluster (*class in pyanp.anp*), 26
ANPNetwork (*class in pyanp.anp*), 27
ANPNode (*class in pyanp.anp*), 32
apply() (*pyanp.prioritizer.PriorityType method*), 18

B

best_std_word_evaluator() (*in module pyanp.rating*), 25

C

calcp0() (*in module pyanp.rowsens*), 18
calculus() (*in module pyanp.limitmatrix*), 12
clean_name() (*in module pyanp.anp*), 33
clean_word() (*in module pyanp.rating*), 25
cluster_connect() (*pyanp.anp.ANPCluster method*), 26
cluster_incon_std_df() (*pyanp.anp.ANPNetwork method*), 27
cluster_names() (*pyanp.anp.ANPNetwork method*), 27
cluster_obj() (*pyanp.anp.ANPNetwork method*), 27
cluster_objs() (*pyanp.anp.ANPNetwork method*), 28
cluster_prioritizer() (*pyanp.anp.ANPNetwork method*), 28

D

data_names() (*pyanp.anp.ANPCluster method*), 26
data_names() (*pyanp.anp.ANPNetwork method*), 28
data_names() (*pyanp.anp.ANPNode method*), 32
data_names() (*pyanp.pairwise.Pairwise method*), 6
data_names() (*pyanp.prioritizer.Prioritizer method*), 17
Direct (*class in pyanp.direct*), 16

E

eval() (*pyanp.rating.WordEval method*), 25

G

geom_avg() (*in module pyanp.priority*), 8
geom_avg_mat() (*in module pyanp.priority*), 8
geom_avg_mats() (*in module pyanp.pairwise*), 8
get_item() (*in module pyanp.anp*), 33
get_key() (*pyanp.rating.WordEval method*), 25
get_matrix() (*in module pyanp.general*), 15
get_node() (*pyanp.ahptree.AHPTree method*), 2
get_node_prioritizer() (*pyanp.anp.ANPNode method*), 32
get_nodes_hash() (*pyanp.ahptree.AHPTree method*), 2
get_unscaled_column() (*pyanp.anp.ANPNode method*), 32
global_priority() (*pyanp.ahptree.AHPTree method*), 3
global_priority() (*pyanp.anp.ANPNetwork method*), 28
global_priority_df() (*pyanp.anp.ANPNetwork method*), 28
global_priority_table() (*pyanp.ahptree.AHPTree method*), 3

H

harker_fix() (*in module pyanp.priority*), 9
has_subnet() (*pyanp.anp.ANPNetwork method*), 28
hiearhcy_formula() (*in module pyanp.limitmatrix*), 12
hierarchy_nodes() (*in module pyanp.limitmatrix*), 12

I

IDEALIZE (*pyanp.prioritizer.PriorityType attribute*), 18
import_pw_series() (*pyanp.anp.ANPNetwork method*), 28
import_rating_series() (*pyanp.anp.ANPNetwork method*), 28
incon_gci() (*in module pyanp.priority*), 9
incon_std() (*in module pyanp.priority*), 9
incon_std() (*pyanp.ahptree.AHPTree method*), 3
incon_std() (*pyanp.pairwise.Pairwise method*), 6
incon_std_series() (*pyanp.ahptree.AHPTree method*), 3
incond_std_table() (*pyanp.ahptree.AHPTree method*), 3
inconsistency_divisor() (*in module pyanp.priority*), 9
influence_fixed() (*in module pyanp.rowsens*), 18
influence_limit() (*in module pyanp.rowsens*), 19
influence_marginal() (*in module pyanp.rowsens*), 19
influence_rank() (*in module pyanp.rowsens*), 20
influence_table() (*in module pyanp.rowsens*), 21
influence_table_plot() (*in module pyanp.rowsens*), 21
invert_priority() (*pyanp.anp.ANPNetwork method*), 28
is_alt() (*pyanp.anp.ANPNetwork method*), 28
is_alt() (*pyanp.pairwise.Pairwise method*), 6
is_alt() (*pyanp.rating.Rating method*), 24
is_node() (*pyanp.anp.ANPCluster method*), 26
is_node_cluster_connection() (*pyanp.anp.ANPNode method*), 32
is_node_node_connection() (*pyanp.anp.ANPNode method*), 32
is_pw_col_name() (*in module pyanp.anp*), 34
is_rating_col_name() (*in module pyanp.anp*), 34
is_user() (*pyanp.anp.ANPNetwork method*), 29
is_user() (*pyanp.pairwise.Pairwise method*), 6
is_user() (*pyanp.rating.Rating method*), 24
isalt() (*pyanp.ahptree.AHPTree method*), 3
islist() (*in module pyanp.general*), 15

K

keys_match_score() (*pyanp.rating.WordEval method*), 25

L

limit_matrix() (*pyanp.anp.ANPNetwork method*), 29
limit_newhierarchy() (*in module pyanp.limitmatrix*), 12
limit_sinks() (*in module pyanp.limitmatrix*), 12
linear_interpolate() (*in module pyanp.general*), 15

M

mat_gci() (*in module pyanp.priority*), 9
matrix() (*pyanp.pairwise.Pairwise method*), 6
matrix_as_df() (*in module pyanp.general*), 15
matrix_matching() (*in module pyanp.exmats*), 14

N

nalts() (*pyanp.ahptree.AHPTree method*), 4
nalts() (*pyanp.pairwise.Pairwise method*), 6
nalts() (*pyanp.prioritizer.Prioritizer method*), 17
nalts() (*pyanp.rating.Rating method*), 24
nclusters() (*pyanp.anp.ANPNetwork method*), 29
nnodes() (*pyanp.anp.ANPCluster method*), 26
nnodes() (*pyanp.anp.ANPNetwork method*), 29
node_connect() (*pyanp.anp.ANPNetwork method*), 29
node_connect() (*pyanp.anp.ANPNode method*), 32
node_connection_matrix() (*pyanp.anp.ANPNetwork method*), 29

node_connections() (*pyanp.anp.ANPNetwork method*), 29
node_incon_std_df() (*pyanp.anp.ANPNetwork method*), 29
node_invert() (*pyanp.anp.ANPNetwork method*), 30
node_names() (*pyanp.anp.ANPCluster method*), 26
node_names() (*pyanp.anp.ANPNetwork method*), 30
node_obj() (*pyanp.anp.ANPCluster method*), 26
node_obj() (*pyanp.anp.ANPNetwork method*), 30
node_objs() (*pyanp.anp.ANPCluster method*), 26
node_objs() (*pyanp.anp.ANPNetwork method*), 30
node_objs_with_subnet() (*pyanp.anp.ANPNetwork method*), 30
node_prioritizer() (*pyanp.anp.ANPNetwork method*), 30
node_pwmatrix() (*pyanp.ahptree.AHPTree method*), 4
nodenames() (*pyanp.ahptree.AHPTree method*), 4
nodepw() (*pyanp.ahptree.AHPTree method*), 4
nodes() (*pyanp.ahptree.AHPTree method*), 4
NORMALIZE (*pyanp.prioritizer.PriorityType attribute*), 18
normalize() (*in module pyanp.limitmatrix*), 13
normalize_cols_dist() (*in module pyanp.limitmatrix*), 13
nusers() (*pyanp.anp.ANPNetwork method*), 30
nusers() (*pyanp.rating.Rating method*), 24

P

p0mode_is_direct() (*in module pyanp.rowsens*), 22
p0mode_is_smart() (*in module pyanp.rowsens*), 22
p0mode_name() (*in module pyanp.rowsens*), 22
Pairwise (*class in pyanp.pairwise*), 5
pairwisematrix_ex() (*in module pyanp.exmats*), 14
prerr_euclidratio() (*in module pyanp.priority*), 9
prerr_ratio_avg() (*in module pyanp.priority*), 10
prerr_ratio_prod() (*in module pyanp.priority*), 10
pri_eigen() (*in module pyanp.priority*), 10
pri_expeigen() (*in module pyanp.priority*), 10
pri_geomavg() (*in module pyanp.priority*), 11
pri_llsm() (*in module pyanp.priority*), 11
Prioritizer (*class in pyanp.prioritizer*), 17
priority() (*in module pyanp.limitmatrix*), 13
priority() (*pyanp.ahptree.AHPTree method*), 5
priority() (*pyanp.anp.ANPNetwork method*), 30
priority() (*pyanp.direct.Direct method*), 16
priority() (*pyanp.pairwise.Pairwise method*), 7
priority() (*pyanp.prioritizer.Prioritizer method*), 17
priority() (*pyanp.rating.Rating method*), 24
priority_df() (*pyanp.prioritizer.Prioritizer method*), 17
priority_from_limit() (*in module pyanp.limitmatrix*), 13
priority_table() (*pyanp.ahptree.AHPTree method*), 5
PriorityType (*class in pyanp.prioritizer*), 18
pyanp.anp (*module*), 26
pyanp.exmats (*module*), 14
pyanp.general (*module*), 15
pyanp.limitmatrix (*module*), 11
pyanp.pairwise (*module*), 5
pyanp.priority (*module*), 8
pyanp.rating (*module*), 23
pyanp.rowsens (*module*), 18

R

rank_change() (*in module pyanp.rowsens*), 22
Rating (*class in pyanp.rating*), 23
ratio_greater_1() (*in module pyanp.priority*), 11
ratio_mat() (*in module pyanp.priority*), 11
RAW (*pyanp.prioritizer.PriorityType attribute*), 18
row_adjust() (*in module pyanp.rowsens*), 22
row_adjust_priority() (*in module pyanp.rowsens*), 23

S

scaled_supermatrix() (*pyanp.anp.ANPNetwork method*), 31
set_alts_cluster() (*pyanp.anp.ANPNetwork method*), 31
set_node_prioritizer_type() (*pyanp.anp.ANPNode method*), 33
set_pairwise_from_supermatrix() (*pyanp.anp.ANPNetwork method*), 31
set_prioritizer_type() (*pyanp.anp.ANPCluster method*), 26
set_word_eval() (*pyanp.rating.Rating method*), 24
size_array_like() (*in module pyanp.priority*), 11
structure_priority() (*pyanp.anp.ANPNetwork method*), 31
subnet() (*pyanp.anp.ANPNetwork method*), 31
subnet_synthesize() (*pyanp.anp.ANPNetwork method*), 31
sum_subnetwork_formula() (*in module pyanp.anp*), 34
supermatrix_ex() (*in module pyanp.exmats*), 15
synthesize() (*pyanp.ahptree.AHPTree method*), 5
synthesize_combine() (*pyanp.anp.ANPNetwork method*), 31

T

two_two_breakdown() (*in module pyanp.limitmatrix*), 14

U

unscaled_supermatrix() (*pyanp.anp.ANPNetwork method*), 31
unvote() (*pyanp.pairwise.Pairwise method*), 7
unwrap_list() (*in module pyanp.general*), 16
user_names() (*pyanp.anp.ANPNetwork method*), 32
user_names() (*pyanp.prioritizer.Prioritizer method*), 17
user_names() (*pyanp.rating.Rating method*), 24
usernames() (*pyanp.ahptree.AHPTree method*), 5
usernames() (*pyanp.direct.Direct method*), 16
usernames() (*pyanp.pairwise.Pairwise method*), 7
utmrowlist_to_npmatrix() (*in module pyanp.priority*), 11

V

vote() (*pyanp.pairwise.Pairwise method*), 7
vote_column() (*pyanp.rating.Rating method*), 24
vote_matrix() (*pyanp.pairwise.Pairwise method*), 7
vote_series() (*pyanp.pairwise.Pairwise method*), 8
vote_values() (*pyanp.rating.Rating method*), 24

W

WordEval (*class in pyanp.rating*), 25
WordEvalType (*class in pyanp.rating*), 25

Z

zero_cols() (*in module pyanp.limitmatrix*), 14