```
"""
========================================================
Hierarchical clustering (:mod:`scipy.cluster.hierarchy`)
========================================================

.. currentmodule:: scipy.cluster.hierarchy

These functions cut hierarchical clusterings into flat clusterings
or find the roots of the forest formed by a cut by providing the flat
cluster ids of each observation.

.. autosummary::
   :toctree: generated/

   fcluster
   fclusterdata
   leaders

These are routines for agglomerative clustering.

.. autosummary::
   :toctree: generated/

   linkage
   single
   complete
   average
   weighted
   centroid
   median
   ward

These routines compute statistics on hierarchies.

.. autosummary::
   :toctree: generated/

   cophenet
   from_mlab_linkage
   inconsistent
   maxinconsts
   maxdists
   maxRstat
   to_mlab_linkage

Routines for visualizing flat clusters.

.. autosummary::
   :toctree: generated/

   dendrogram

These are data structures and routines for representing hierarchies as
tree objects.

.. autosummary::
   :toctree: generated/

   ClusterNode
   leaves_list
   to_tree
   cut_tree
   optimal_leaf_ordering
```

```
These are predicates for checking the validity of linkage and
inconsistency matrices as well as for checking isomorphism of two
flat cluster assignments.

.. autosummary::
   :toctree: generated/

   is_valid_im
   is_valid_linkage
   is_isomorphic
   is_monotonic
   correspond
   num_obs_linkage

Utility routines for plotting:

.. autosummary::
   :toctree: generated/

   set_link_color_palette

References
----------

.. [1] "Statistics toolbox." API Reference Documentation. The MathWorks.
   http://www.mathworks.com/access/helpdesk/help/toolbox/stats/.
   Accessed October 1, 2007.

.. [2] "Hierarchical clustering." API Reference Documentation.
   The Wolfram Research, Inc.
   https://reference.wolfram.com/language/HierarchicalClustering/tutorial/
   HierarchicalClustering.html.
   Accessed October 1, 2007.

.. [3] Gower, JC and Ross, GJS. "Minimum Spanning Trees and Single Linkage
   Cluster Analysis." Applied Statistics. 18(1): pp. 54--64. 1969.

.. [4] Ward Jr, JH. "Hierarchical grouping to optimize an objective
   function." Journal of the American Statistical Association. 58(301):
   pp. 236--44. 1963.

.. [5] Johnson, SC. "Hierarchical clustering schemes." Psychometrika.
   32(2): pp. 241--54. 1966.

.. [6] Sneath, PH and Sokal, RR. "Numerical taxonomy." Nature. 193: pp.
   855--60. 1962.

.. [7] Batagelj, V. "Comparing resemblance measures." Journal of
   Classification. 12: pp. 73--90. 1995.

.. [8] Sokal, RR and Michener, CD. "A statistical method for evaluating
   systematic relationships." Scientific Bulletins. 38(22):
   pp. 1409--38. 1958.

.. [9] Edelbrock, C. "Mixture model tests of hierarchical clustering
   algorithms: the problem of classifying everybody." Multivariate
   Behavioral Research. 14: pp. 367--84. 1979.

.. [10] Jain, A., and Dubes, R., "Algorithms for Clustering Data."
   Prentice-Hall. Englewood Cliffs, NJ. 1988.

.. [11] Fisher, RA "The use of multiple measurements in taxonomic
   problems." Annals of Eugenics, 7(2): 179-188. 1936
```

```
* MATLAB and MathWorks are registered trademarks of The MathWorks, Inc.

* Mathematica is a registered trademark of The Wolfram Research, Inc.

"""
from __future__ import division, print_function, absolute_import

# Copyright (C) Damian Eads, 2007-2008. New BSD License.

# hierarchy.py (derived from cluster.py, http://scipy-cluster.googlecode.com)
#
# Author: Damian Eads
# Date:   September 22, 2007
#
# Copyright (c) 2007, 2008, Damian Eads
#
# All rights reserved.
#
# Redistribution and use in source and binary forms, with or without
# modification, are permitted provided that the following conditions
# are met:
#   - Redistributions of source code must retain the above
#     copyright notice, this list of conditions and the
#     following disclaimer.
#   - Redistributions in binary form must reproduce the above copyright
#     notice, this list of conditions and the following disclaimer
#     in the documentation and/or other materials provided with the
#     distribution.
#   - Neither the name of the author nor the names of its
#     contributors may be used to endorse or promote products derived
#     from this software without specific prior written permission.
#
# THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
# "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
# LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
# A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
# OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
# SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
# LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
# DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
# THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
# (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
# OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

# Changes to this code were made by Ed Egan, 2022. Copyleft.
# Addition 1: lines 188 to 207
# Addition 1: lines 734 to 743

import warnings
import bisect
from collections import deque

import numpy as np
from . import _hierarchy, _optimal_leaf_ordering
import scipy.spatial.distance as distance

from scipy._lib.six import string_types
from scipy._lib.six import xrange

################## Begin Addition (section 1) ##################

_DISTANCE_FILENAME = r'E:\projects\hca\CBSARandom6Locs2006Lookup.txt' #Should
be passed in.
```

```python
_DISTANCES={}
with open(_DISTANCE_FILENAME) as f:
    next(f) #loose the header row
    for line in f:
        line=line.rstrip('\r\n')
        parts = line.split('\t')
        source=(float(parts[0]),float(parts[1]))
        dest=(float(parts[2]),float(parts[3]))
        dist=parts[4]
        if source in _DISTANCES:
            _DISTANCES[source][dest]=dist
        else:
            _DISTANCES[source]={}
            _DISTANCES[source][dest]=dist

################## End Addition (section 1) ##################

_LINKAGE_METHODS = {'single': 0, 'complete': 1, 'average': 2, 'centroid': 3,
                    'median': 4, 'ward': 5, 'weighted': 6}
_EUCLIDEAN_METHODS = ('centroid', 'median', 'ward')

__all__ = ['ClusterNode', 'average', 'centroid', 'complete', 'cophenet',
           'correspond', 'cut_tree', 'dendrogram', 'fcluster', 'fclusterdata',
           'from_mlab_linkage', 'inconsistent', 'is_isomorphic',
           'is_monotonic', 'is_valid_im', 'is_valid_linkage', 'leaders',
           'leaves_list', 'linkage', 'maxRstat', 'maxdists', 'maxinconsts',
           'median', 'num_obs_linkage', 'optimal_leaf_ordering',
           'set_link_color_palette', 'single', 'to_mlab_linkage', 'to_tree',
           'ward', 'weighted', 'distance']

class ClusterWarning(UserWarning):
    pass

def _warning(s):
    warnings.warn('scipy.cluster: %s' % s, ClusterWarning, stacklevel=3)


def _copy_array_if_base_present(a):
    """
    Copy the array if its base points to a parent array.
    """
    if a.base is not None:
        return a.copy()
    elif np.issubsctype(a, np.float32):
        return np.array(a, dtype=np.double)
    else:
        return a


def _copy_arrays_if_base_present(T):
    """
    Accept a tuple of arrays T. Copies the array T[i] if its base array
    points to an actual array. Otherwise, the reference is just copied.
    This is useful if the arrays are being passed to a C function that
    does not do proper striding.
    """
    l = [_copy_array_if_base_present(a) for a in T]
    return l


def _randdm(pnts):
    """
    Generate a random distance matrix stored in condensed form.
    Parameters
    ----------
    pnts : int
        The number of points in the distance matrix. Has to be at least 2.

    Returns
    -------
    D : ndarray
        A ``pnts * (pnts - 1) / 2`` sized vector is returned.
    """
    if pnts >= 2:
        D = np.random.rand(pnts * (pnts - 1) / 2)
    else:
        raise ValueError("The number of points in the distance matrix "
                         "must be at least 2.")
    return D


def single(y):
    """
    Perform single/min/nearest linkage on the condensed distance matrix ``y``.

    Parameters
    ----------
    y : ndarray
        The upper triangular of the distance matrix. The result of
        ``pdist`` is returned in this form.

    Returns
    -------
    Z : ndarray
        The linkage matrix.

    See Also
    --------
    linkage: for advanced creation of hierarchical clusterings.
    scipy.spatial.distance.pdist : pairwise distance metrics

    """
    return linkage(y, method='single', metric='euclidean')


def complete(y):
    """
    Perform complete/max/farthest point linkage on a condensed distance
    matrix.

    Parameters
    ----------
    y : ndarray
        The upper triangular of the distance matrix. The result of
        ``pdist`` is returned in this form.

    Returns
    -------
    Z : ndarray
        A linkage matrix containing the hierarchical clustering. See
        the `linkage` function documentation for more information
        on its structure.

    See Also
    --------
    linkage: for advanced creation of hierarchical clusterings.
    scipy.spatial.distance.pdist : pairwise distance metrics

    """
    return linkage(y, method='complete', metric='euclidean')


def average(y):
    """
    Perform average/UPGMA linkage on a condensed distance matrix.

    Parameters
    ----------
    y : ndarray
        The upper triangular of the distance matrix. The result of
        ``pdist`` is returned in this form.

    Returns
    -------
    Z : ndarray
        A linkage matrix containing the hierarchical clustering. See
        `linkage` for more information on its structure.

    See Also
    --------
    linkage: for advanced creation of hierarchical clusterings.
    scipy.spatial.distance.pdist : pairwise distance metrics

    """
    return linkage(y, method='average', metric='euclidean')


def weighted(y):
    """
    Perform weighted/WPGMA linkage on the condensed distance matrix.

    See `linkage` for more information on the return
    structure and algorithm.

    Parameters
    ----------
    y : ndarray
        The upper triangular of the distance matrix. The result of
        ``pdist`` is returned in this form.

    Returns
    -------
    Z : ndarray
        A linkage matrix containing the hierarchical clustering. See
        `linkage` for more information on its structure.

    See Also
    --------
    linkage : for advanced creation of hierarchical clusterings.
    scipy.spatial.distance.pdist : pairwise distance metrics

    """
    return linkage(y, method='weighted', metric='euclidean')


def centroid(y):
    """
    Perform centroid/UPGMC linkage.

    See `linkage` for more information on the input matrix,
    return structure, and algorithm.

    The following are common calling conventions:

    1. ``Z = centroid(y)``

       Performs centroid/UPGMC linkage on the condensed distance
       matrix ``y``.

    2. ``Z = centroid(X)``

       Performs centroid/UPGMC linkage on the observation matrix ``X``
       using Euclidean distance as the distance metric.

    Parameters
    ----------
    y : ndarray
        A condensed distance matrix. A condensed
        distance matrix is a flat array containing the upper
        triangular of the distance matrix. This is the form that
        ``pdist`` returns. Alternatively, a collection of
        m observation vectors in n dimensions may be passed as
        a m by n array.

    Returns
    -------
    Z : ndarray
        A linkage matrix containing the hierarchical clustering. See
        the `linkage` function documentation for more information
        on its structure.

    See Also
    --------
    linkage: for advanced creation of hierarchical clusterings.

    """
    return linkage(y, method='centroid', metric='euclidean')


def median(y):
    """
    Perform median/WPGMC linkage.

    See `linkage` for more information on the return structure
    and algorithm.

     The following are common calling conventions:

     1. ``Z = median(y)``

        Performs median/WPGMC linkage on the condensed distance matrix
        ``y``.  See ``linkage`` for more information on the return
        structure and algorithm.

     2. ``Z = median(X)``

        Performs median/WPGMC linkage on the observation matrix ``X``
        using Euclidean distance as the distance metric. See `linkage`
        for more information on the return structure and algorithm.

    Parameters
    ----------
    y : ndarray
        A condensed distance matrix. A condensed
        distance matrix is a flat array containing the upper
        triangular of the distance matrix. This is the form that
        ``pdist`` returns.  Alternatively, a collection of
        m observation vectors in n dimensions may be passed as
        a m by n array.

    Returns
    -------
    Z : ndarray
        The hierarchical clustering encoded as a linkage matrix.

    See Also
    --------
    linkage: for advanced creation of hierarchical clusterings.
    scipy.spatial.distance.pdist : pairwise distance metrics

    """
    return linkage(y, method='median', metric='euclidean')


def ward(y):
    """
    Perform Ward's linkage on a condensed distance matrix.

    See `linkage` for more information on the return structure
    and algorithm.

    The following are common calling conventions:

    1. ``Z = ward(y)``
       Performs Ward's linkage on the condensed distance matrix ``y``.

    2. ``Z = ward(X)``
       Performs Ward's linkage on the observation matrix ``X`` using
       Euclidean distance as the distance metric.

    Parameters
    ----------
    y : ndarray
        A condensed distance matrix. A condensed
        distance matrix is a flat array containing the upper
        triangular of the distance matrix. This is the form that
        ``pdist`` returns.  Alternatively, a collection of
        m observation vectors in n dimensions may be passed as
        a m by n array.

    Returns
    -------
    Z : ndarray
        The hierarchical clustering encoded as a linkage matrix. See
        `linkage` for more information on the return structure and
        algorithm.

    See Also
    --------
    linkage: for advanced creation of hierarchical clusterings.
    scipy.spatial.distance.pdist : pairwise distance metrics

    """
    return linkage(y, method='ward', metric='euclidean')


def linkage(y, method='single', metric='euclidean', optimal_ordering=False):
    """
    Perform hierarchical/agglomerative clustering.

    The input y may be either a 1d condensed distance matrix
    or a 2d array of observation vectors.

    If y is a 1d condensed distance matrix,
    then y must be a :math:`\\binom{n}{2}` sized
    vector where n is the number of original observations paired
    in the distance matrix. The behavior of this function is very
    similar to the MATLAB linkage function.

    A :math:`(n-1)` by 4 matrix ``Z`` is returned. At the
    :math:`i`-th iteration, clusters with indices ``Z[i, 0]`` and
    ``Z[i, 1]`` are combined to form cluster :math:`n + i`. A
    cluster with an index less than :math:`n` corresponds to one of
    the :math:`n` original observations. The distance between
    clusters ``Z[i, 0]`` and ``Z[i, 1]`` is given by ``Z[i, 2]``. The
    fourth value ``Z[i, 3]`` represents the number of original
    observations in the newly formed cluster.

    The following linkage methods are used to compute the distance
    :math:`d(s, t)` between two clusters :math:`s` and
    :math:`t`. The algorithm begins with a forest of clusters that
    have yet to be used in the hierarchy being formed. When two
    clusters :math:`s` and :math:`t` from this forest are combined
    into a single cluster :math:`u`, :math:`s` and :math:`t` are
    removed from the forest, and :math:`u` is added to the
    forest. When only one cluster remains in the forest, the algorithm
    stops, and this cluster becomes the root.

    A distance matrix is maintained at each iteration. The ``d[i,j]``
    entry corresponds to the distance between cluster :math:`i` and
    :math:`j` in the original forest.

    At each iteration, the algorithm must update the distance matrix
    to reflect the distance of the newly formed cluster u with the
    remaining clusters in the forest.

    Suppose there are :math:`|u|` original observations
    :math:`u[0], \\ldots, u[|u|-1]` in cluster :math:`u` and
    :math:`|v|` original objects :math:`v[0], \\ldots, v[|v|-1]` in
    cluster :math:`v`. Recall :math:`s` and :math:`t` are
    combined to form cluster :math:`u`. Let :math:`v` be any
    remaining cluster in the forest that is not :math:`u`.

    The following are methods for calculating the distance between the
    newly formed cluster :math:`u` and each :math:`v`.

      * method='single' assigns

        .. math::
           d(u,v) = \\min(dist(u[i],v[j]))

        for all points :math:`i` in cluster :math:`u` and
        :math:`j` in cluster :math:`v`. This is also known as the
        Nearest Point Algorithm.

      * method='complete' assigns

        .. math::
           d(u, v) = \\max(dist(u[i],v[j]))

        for all points :math:`i` in cluster u and :math:`j` in
        cluster :math:`v`. This is also known by the Farthest Point
        Algorithm or Voor Hees Algorithm.

      * method='average' assigns

        .. math::
           d(u,v) = \\sum_{ij} \\frac{d(u[i], v[j])}
                                   {(|u|*|v|)}

        for all points :math:`i` and :math:`j` where :math:`|u|`
        and :math:`|v|` are the cardinalities of clusters :math:`u`
        and :math:`v`, respectively. This is also called the UPGMA
        algorithm.

      * method='weighted' assigns

        .. math::
           d(u,v) = (dist(s,v) + dist(t,v))/2

        where cluster u was formed with cluster s and t and v
        is a remaining cluster in the forest. (also called WPGMA)

      * method='centroid' assigns

        .. math::
           dist(s,t) = ||c_s-c_t||_2

        where :math:`c_s` and :math:`c_t` are the centroids of
        clusters :math:`s` and :math:`t`, respectively. When two
        clusters :math:`s` and :math:`t` are combined into a new
        cluster :math:`u`, the new centroid is computed over all the
        original objects in clusters :math:`s` and :math:`t`. The
        distance then becomes the Euclidean distance between the
        centroid of :math:`u` and the centroid of a remaining cluster
        :math:`v` in the forest. This is also known as the UPGMC
        algorithm.

      * method='median' assigns :math:`d(s,t)` like the ``centroid``
        method. When two clusters :math:`s` and :math:`t` are combined
        into a new cluster :math:`u`, the average of centroids s and t
        give the new centroid :math:`u`. This is also known as the
        WPGMC algorithm.

      * method='ward' uses the Ward variance minimization algorithm.
        The new entry :math:`d(u,v)` is computed as follows,

        .. math::

           d(u,v) = \\sqrt{\\frac{|v|+|s|}
                               {T}d(v,s)^2
                          + \\frac{|v|+|t|}
                               {T}d(v,t)^2
                          - \\frac{|v|}
                               {T}d(s,t)^2}

        where :math:`u` is the newly joined cluster consisting of
        clusters :math:`s` and :math:`t`, :math:`v` is an unused
        cluster in the forest, :math:`T=|v|+|s|+|t|`, and
        :math:`|*|` is the cardinality of its argument. This is also
        known as the incremental algorithm.

    Warning: When the minimum distance pair in the forest is chosen, there
    may be two or more pairs with the same minimum distance. This
    implementation may choose a different minimum than the MATLAB
    version.
    Parameters
    ----------
    y : ndarray
        A condensed distance matrix. A condensed distance matrix
        is a flat array containing the upper triangular of the distance
        matrix.
        This is the form that ``pdist`` returns. Alternatively, a collection
        of
        :math:`m` observation vectors in :math:`n` dimensions may be passed as
        an :math:`m` by :math:`n` array. All elements of the condensed
        distance
        matrix must be finite, i.e. no NaNs or infs.
    method : str, optional
        The linkage algorithm to use. See the ``Linkage Methods`` section
        below
        for full descriptions.
    metric : str or function, optional
        The distance metric to use in the case that y is a collection of
        observation vectors; ignored otherwise. See the ``pdist``
        function for a list of valid distance metrics. A custom distance
        function can also be used.
    optimal_ordering : bool, optional
        If True, the linkage matrix will be reordered so that the distance
        between successive leaves is minimal. This results in a more intuitive
        tree structure when the data are visualized. defaults to False,
        because
        this algorithm can be slow, particularly on large datasets [2]_. See
        also the `optimal_leaf_ordering` function.

        .. versionadded:: 1.0.0

    Returns
    -------
    Z : ndarray
        The hierarchical clustering encoded as a linkage matrix.

    Notes
    -----
    1. For method 'single' an optimized algorithm based on minimum spanning
       tree is implemented. It has time complexity :math:`O(n^2)`.
       For methods 'complete', 'average', 'weighted' and 'ward' an algorithm
       called nearest-neighbors chain is implemented. It also has time
       complexity :math:`O(n^2)`.
       For other methods a naive algorithm is implemented with :math:`O(n^3)`
       time complexity.
       All algorithms use :math:`O(n^2)` memory.
       Refer to [1]_ for details about the algorithms.
    2. Methods 'centroid', 'median' and 'ward' are correctly defined only if
       Euclidean pairwise metric is used. If `y` is passed as precomputed
       pairwise distances, then it is a user responsibility to assure that
       these distances are in fact Euclidean, otherwise the produced result
       will be incorrect.

    See Also
    --------
    scipy.spatial.distance.pdist : pairwise distance metrics

    References
    ----------
    .. [1] Daniel Mullner, "Modern hierarchical, agglomerative clustering
           algorithms", :arXiv:`1109.2378v1`.
    .. [2] Ziv Bar-Joseph, David K. Gifford, Tommi S. Jaakkola, "Fast optimal
           leaf ordering for hierarchical clustering", 2001. Bioinformatics
           https://doi.org/10.1093/bioinformatics/17.suppl_1.S22

    Examples
    --------
    >>> from scipy.cluster.hierarchy import dendrogram, linkage
    >>> from matplotlib import pyplot as plt
    >>> X = [[i] for i in [2, 8, 0, 4, 1, 9, 9, 0]]

    >>> Z = linkage(X, 'ward')
    >>> fig = plt.figure(figsize=(25, 10))
    >>> dn = dendrogram(Z)

    >>> Z = linkage(X, 'single')
    >>> fig = plt.figure(figsize=(25, 10))
    >>> dn = dendrogram(Z)
    >>> plt.show()
    """
    if method not in _LINKAGE_METHODS:
        raise ValueError("Invalid method: {0}".format(method))

    y = _convert_to_double(np.asarray(y, order='c'))

    if y.ndim == 1:
        distance.is_valid_y(y, throw=True, name='y')
        [y] = _copy_arrays_if_base_present([y])
    elif y.ndim == 2:
        if method in _EUCLIDEAN_METHODS and metric != 'euclidean':
            raise ValueError("Method '{0}' requires the distance metric "
                             "to be Euclidean".format(method))
        if y.shape[0] == y.shape[1] and np.allclose(np.diag(y), 0):
            if np.all(y >= 0) and np.allclose(y, y.T):
                _warning('The symmetric non-negative hollow observation '
                         'matrix looks suspiciously like an uncondensed '
                         'distance matrix')

        #y = distance.pdist(y, metric)

################## Begin Addition (section 2) ##################

        points=[tuple(row) for row in y]
        sqform=np.zeros((len(points),len(points)))
        for i,source in enumerate(points):
            for j,dest in enumerate(points):
                sqform[i,j]=_DISTANCES[source][dest]
        y=distance.squareform(sqform)

################## End Addition (section 2) ##################

    else:
        raise ValueError("`y` must be 1 or 2 dimensional.")

    if not np.all(np.isfinite(y)):
        raise ValueError("The condensed distance matrix must contain only "
                         "finite values.")

    n = int(distance.num_obs_y(y))
    method_code = _LINKAGE_METHODS[method]

    if method == 'single':
        result = _hierarchy.mst_single_linkage(y, n)
    elif method in ['complete', 'average', 'weighted', 'ward']:
        result = _hierarchy.nn_chain(y, n, method_code)
    else:
        result = _hierarchy.fast_linkage(y, n, method_code)
    if optimal_ordering:
        return optimal_leaf_ordering(result, y)
    else:
        return result


class ClusterNode:
    """
    A tree node class for representing a cluster.

    Leaf nodes correspond to original observations, while non-leaf nodes
    correspond to non-singleton clusters.

    The `to_tree` function converts a matrix returned by the linkage
    function into an easy-to-use tree representation.

    All parameter names are also attributes.

    Parameters
    ----------
    id : int
        The node id.
    left : ClusterNode instance, optional
        The left child tree node.
    right : ClusterNode instance, optional
        The right child tree node.
    dist : float, optional
        Distance for this cluster in the linkage matrix.
    count : int, optional
        The number of samples in this cluster.

    See Also
    --------
    to_tree : for converting a linkage matrix ``Z`` into a tree object.

    """

    def __init__(self, id, left=None, right=None, dist=0, count=1):
        if id < 0:
            raise ValueError('The id must be non-negative.')
        if dist < 0:
            raise ValueError('The distance must be non-negative.')
        if (left is None and right is not None) or \
           (left is not None and right is None):
            raise ValueError('Only full or proper binary trees are permitted.'
                             '  This node has one child.')
        if count < 1:
            raise ValueError('A cluster must contain at least one original '
                             'observation.')
        self.id = id
        self.left = left
        self.right = right
        self.dist = dist
        if self.left is None:
            self.count = count
        else:
            self.count = left.count + right.count

    def __lt__(self, node):
        if not isinstance(node, ClusterNode):
            raise ValueError("Can't compare ClusterNode "
                             "to type {}".format(type(node)))
        return self.dist < node.dist
    def __gt__(self, node):
        if not isinstance(node, ClusterNode):
            raise ValueError("Can't compare ClusterNode "
                             "to type {}".format(type(node)))
        return self.dist > node.dist

    def __eq__(self, node):
        if not isinstance(node, ClusterNode):
            raise ValueError("Can't compare ClusterNode "
                             "to type {}".format(type(node)))
        return self.dist == node.dist

    def get_id(self):
        """
        The identifier of the target node.

        For ``0 <= i < n``, `i` corresponds to original observation i.
        For ``n <= i < 2n-1``, `i` corresponds to non-singleton cluster formed
        at iteration ``i-n``.

        Returns
        -------
        id : int
            The identifier of the target node.

        """
        return self.id

    def get_count(self):
        """
        The number of leaf nodes (original observations) belonging to
        the cluster node nd. If the target node is a leaf, 1 is
        returned.

        Returns
        -------
        get_count : int
            The number of leaf nodes below the target node.

        """
        return self.count

    def get_left(self):
        """
        Return a reference to the left child tree object.

        Returns
        -------
        left : ClusterNode
            The left child of the target node.  If the node is a leaf,
            None is returned.

        """
        return self.left

    def get_right(self):
        """
        Return a reference to the right child tree object.

        Returns
        -------
        right : ClusterNode
            The left child of the target node.  If the node is a leaf,
            None is returned.

        """
        return self.right

    def is_leaf(self):
        """
        Return True if the target node is a leaf.

        Returns
        -------
        leafness : bool
            True if the target node is a leaf node.

        """
        return self.left is None

    def pre_order(self, func=(lambda x: x.id)):
        """
        Perform pre-order traversal without recursive function calls.

        When a leaf node is first encountered, ``func`` is called with
        the leaf node as its argument, and its result is appended to
        the list.

        For example, the statement::

           ids = root.pre_order(lambda x: x.id)

        returns a list of the node ids corresponding to the leaf nodes
        of the tree as they appear from left to right.

        Parameters
        ----------
        func : function
            Applied to each leaf ClusterNode object in the pre-order
            traversal.
            Given the ``i``-th leaf node in the pre-order traversal ``n[i]``,
            the result of ``func(n[i])`` is stored in ``L[i]``. If not
            provided, the index of the original observation to which the node
            corresponds is used.

        Returns
        -------
        L : list
            The pre-order traversal.

        """
        # Do a preorder traversal, caching the result. To avoid having to do
        # recursion, we'll store the previous index we've visited in a vector.
        n = self.count

        curNode = [None] * (2 * n)
        lvisited = set()
        rvisited = set()
        curNode[0] = self
        k = 0
        preorder = []
        while k >= 0:
            nd = curNode[k]
            ndid = nd.id
            if nd.is_leaf():
                preorder.append(func(nd))
                k = k - 1
            else:
                if ndid not in lvisited:
                    curNode[k + 1] = nd.left
                    lvisited.add(ndid)
                    k = k + 1
                elif ndid not in rvisited:
                    curNode[k + 1] = nd.right
                    rvisited.add(ndid)
                    k = k + 1
                # If we've visited the left and right of this non-leaf
                # node already, go up in the tree.
                else:
                    k = k - 1

        return preorder


_cnode_bare = ClusterNode(0)
_cnode_type = type(ClusterNode)


def _order_cluster_tree(Z):
    """
    Return clustering nodes in bottom-up order by distance.

    Parameters
    ----------
    Z : scipy.cluster.linkage array
        The linkage matrix.

    Returns
    -------
    nodes : list
        A list of ClusterNode objects.
    """
    q = deque()
    tree = to_tree(Z)
    q.append(tree)
    nodes = []

    while q:
        node = q.popleft()
        if not node.is_leaf():
            bisect.insort_left(nodes, node)
            q.append(node.get_right())
            q.append(node.get_left())
    return nodes


def cut_tree(Z, n_clusters=None, height=None):
    """
    Given a linkage matrix Z, return the cut tree.

    Parameters
    ----------
    Z : scipy.cluster.linkage array
        The linkage matrix.
    n_clusters : array_like, optional
        Number of clusters in the tree at the cut point.
    height : array_like, optional
        The height at which to cut the tree.  Only possible for ultrametric
        trees.

    Returns
    -------
    cutree : array
        An array indicating group membership at each agglomeration step.
        I.e.,
        for a full cut tree, in the first column each data point is in its own
        cluster.  At the next step, two nodes are merged.  Finally all
        singleton and non-singleton clusters are in one group.  If
        `n_clusters`
        or `height` is given, the columns correspond to the columns of
        `n_clusters` or `height`.

    Examples
    --------
    >>> from scipy import cluster
    >>> np.random.seed(23)
    >>> X = np.random.randn(50, 4)
    >>> Z = cluster.hierarchy.ward(X)
    >>> cutree = cluster.hierarchy.cut_tree(Z, n_clusters=[5, 10])
    >>> cutree[:10]
    array([[0, 0],
           [1, 1],
           [2, 2],
           [3, 3],
           [3, 4],
           [2, 2],
           [0, 0],
           [1, 5],
           [3, 6],
           [4, 7]])

    """
    nobs = num_obs_linkage(Z)
    nodes = _order_cluster_tree(Z)

    if height is not None and n_clusters is not None:
        raise ValueError("At least one of either height or n_clusters "
                         "must be None")
    elif height is None and n_clusters is None:  # return the full cut tree
        cols_idx = np.arange(nobs)
    elif height is not None:
        heights = np.array([x.dist for x in nodes])
        cols_idx = np.searchsorted(heights, height)
    else:
        cols_idx = nobs - np.searchsorted(np.arange(nobs), n_clusters)

    try:
        n_cols = len(cols_idx)
    except TypeError:  # scalar
        n_cols = 1
        cols_idx = np.array([cols_idx])

    groups = np.zeros((n_cols, nobs), dtype=int)
    last_group = np.arange(nobs)
    if 0 in cols_idx:
        groups[0] = last_group

    for i, node in enumerate(nodes):
        idx = node.pre_order()
        this_group = last_group.copy()
        this_group[idx] = last_group[idx].min()
        this_group[this_group > last_group[idx].max()] -= 1
        if i + 1 in cols_idx:
            groups[np.where(i + 1 == cols_idx)[0]] = this_group
        last_group = this_group
    return groups.T


def to_tree(Z, rd=False):
    """
    Convert a linkage matrix into an easy-to-use tree object.

    The reference to the root `ClusterNode` object is returned (by default).

    Each `ClusterNode` object has a ``left``, ``right``, ``dist``, ``id``,
    and ``count`` attribute. The left and right attributes point to
    ClusterNode objects that were combined to generate the cluster.
    If both are None then the `ClusterNode` object is a leaf node, its count
    must be 1, and its distance is meaningless but set to 0.

    *Note: This function is provided for the convenience of the library
    user. ClusterNodes are not used as input to any of the functions in this
    library.*

    Parameters
    ----------
    Z : ndarray
        The linkage matrix in proper form (see the `linkage`
        function documentation).
    rd : bool, optional
        When False (default), a reference to the root `ClusterNode` object is
        returned.  Otherwise, a tuple ``(r, d)`` is returned. ``r`` is a
        reference to the root node while ``d`` is a list of `ClusterNode`
        objects - one per original entry in the linkage matrix plus entries
        for all clustering steps.  If a cluster id is
        less than the number of samples ``n`` in the data that the linkage
        matrix describes, then it corresponds to a singleton cluster (leaf
        node).
        See `linkage` for more information on the assignment of cluster ids
        to clusters.

    Returns
    -------
    tree : ClusterNode or tuple (ClusterNode, list of ClusterNode)
        If ``rd`` is False, a `ClusterNode`.
        If ``rd`` is True, a list of length ``2*n - 1``, with ``n`` the number
        of samples.  See the description of `rd` above for more details.

    See Also
    --------
    linkage, is_valid_linkage, ClusterNode

    Examples
    --------
    >>> from scipy.cluster import hierarchy
    >>> x = np.random.rand(10).reshape(5, 2)
    >>> Z = hierarchy.linkage(x)
    >>> hierarchy.to_tree(Z)
    <scipy.cluster.hierarchy.ClusterNode object at ...
    >>> rootnode, nodelist = hierarchy.to_tree(Z, rd=True)
    >>> rootnode
    <scipy.cluster.hierarchy.ClusterNode object at ...
    >>> len(nodelist)
    9

    """
    Z = np.asarray(Z, order='c')
    is_valid_linkage(Z, throw=True, name='Z')
    # Number of original objects is equal to the number of rows minus 1.
    n = Z.shape[0] + 1

    # Create a list full of None's to store the node objects
    d = [None] * (n * 2 - 1)

    # Create the nodes corresponding to the n original objects.
    for i in xrange(0, n):
        d[i] = ClusterNode(i)

    nd = None

    for i in xrange(0, n - 1):
        fi = int(Z[i, 0])
        fj = int(Z[i, 1])
        if fi > i + n:
            raise ValueError(('Corrupt matrix Z. Index to derivative cluster '
                              'is used before it is formed. See row %d, '
                              'column 0') % fi)
        if fj > i + n:
            raise ValueError(('Corrupt matrix Z. Index to derivative cluster '
                              'is used before it is formed. See row %d, '
                              'column 1') % fj)
        nd = ClusterNode(i + n, d[fi], d[fj], Z[i, 2])
        #                ^ id   ^ left ^ right ^ dist
        if Z[i, 3] != nd.count:
            raise ValueError(('Corrupt matrix Z. The count Z[%d,3] is '
                              'incorrect.') % i)
        d[n + i] = nd

    if rd:
        return (nd, d)
    else:
        return nd


def optimal_leaf_ordering(Z, y, metric='euclidean'):
    """
    Given a linkage matrix Z and distance, reorder the cut tree.

    Parameters
    ----------
    Z : ndarray
        The hierarchical clustering encoded as a linkage matrix. See
        `linkage` for more information on the return structure and
        algorithm.
    y : ndarray
        The condensed distance matrix from which Z was generated.
        Alternatively, a collection of m observation vectors in n
        dimensions may be passed as a m by n array.
    metric : str or function, optional
        The distance metric to use in the case that y is a collection of
        observation vectors; ignored otherwise. See the ``pdist``
        function for a list of valid distance metrics. A custom distance
        function can also be used.

    Returns
    -------
    Z_ordered : ndarray
        A copy of the linkage matrix Z, reordered to minimize the distance
        between adjacent leaves.

    Examples
    --------
    >>> from scipy.cluster import hierarchy
    >>> np.random.seed(23)
    >>> X = np.random.randn(10,10)
    >>> Z = hierarchy.ward(X)
    >>> hierarchy.leaves_list(Z)
    array([0, 5, 3, 9, 6, 8, 1, 4, 2, 7], dtype=int32)
    >>> hierarchy.leaves_list(hierarchy.optimal_leaf_ordering(Z, X))
    array([3, 9, 0, 5, 8, 2, 7, 4, 1, 6], dtype=int32)

    """
    Z = np.asarray(Z, order='c')
    is_valid_linkage(Z, throw=True, name='Z')

    y = _convert_to_double(np.asarray(y, order='c'))

    if y.ndim == 1:
        distance.is_valid_y(y, throw=True, name='y')
        [y] = _copy_arrays_if_base_present([y])
    elif y.ndim == 2:
        if y.shape[0] == y.shape[1] and np.allclose(np.diag(y), 0):
            if np.all(y >= 0) and np.allclose(y, y.T):
                _warning('The symmetric non-negative hollow observation '
                         'matrix looks suspiciously like an uncondensed '
                         'distance matrix')
        y = distance.pdist(y, metric)
    else:
        raise ValueError("`y` must be 1 or 2 dimensional.")

    if not np.all(np.isfinite(y)):
        raise ValueError("The condensed distance matrix must contain only "
                         "finite values.")

    return _optimal_leaf_ordering.optimal_leaf_ordering(Z, y)


def _convert_to_bool(X):
    if X.dtype != bool:
        X = X.astype(bool)
    if not X.flags.contiguous:
        X = X.copy()
    return X


def _convert_to_double(X):
    if X.dtype != np.double:
        X = X.astype(np.double)
    if not X.flags.contiguous:
        X = X.copy()
    return X


def cophenet(Z, Y=None):
    """
    Calculate the cophenetic distances between each observation in
    the hierarchical clustering defined by the linkage ``Z``.

    Suppose ``p`` and ``q`` are original observations in
    disjoint clusters ``s`` and ``t``, respectively and
    ``s`` and ``t`` are joined by a direct parent cluster
    ``u``. The cophenetic distance between observations
    ``i`` and ``j`` is simply the distance between
    clusters ``s`` and ``t``.
    Parameters
    ----------
    Z : ndarray
        The hierarchical clustering encoded as an array
        (see `linkage` function).
    Y : ndarray (optional)
        Calculates the cophenetic correlation coefficient ``c`` of a
        hierarchical clustering defined by the linkage matrix `Z`
        of a set of :math:`n` observations in :math:`m`
        dimensions. `Y` is the condensed distance matrix from which
        `Z` was generated.

    Returns
    -------
    c : ndarray
        The cophentic correlation distance (if ``Y`` is passed).
    d : ndarray
        The cophenetic distance matrix in condensed form. The
        :math:`ij` th entry is the cophenetic distance between
        original observations :math:`i` and :math:`j`.

    """
    Z = np.asarray(Z, order='c')
    is_valid_linkage(Z, throw=True, name='Z')
    Zs = Z.shape
    n = Zs[0] + 1

    zz = np.zeros((n * (n-1)) // 2, dtype=np.double)
    # Since the C code does not support striding using strides.
    # The dimensions are used instead.
    Z = _convert_to_double(Z)

    _hierarchy.cophenetic_distances(Z, zz, int(n))
    if Y is None:
        return zz

    Y = np.asarray(Y, order='c')
    distance.is_valid_y(Y, throw=True, name='Y')

    z = zz.mean()
    y = Y.mean()
    Yy = Y - y
    Zz = zz - z
    numerator = (Yy * Zz)
    denomA = Yy**2
    denomB = Zz**2
    c = numerator.sum() / np.sqrt((denomA.sum() * denomB.sum()))
    return (c, zz)


def inconsistent(Z, d=2):
    r"""
    Calculate inconsistency statistics on a linkage matrix.

    Parameters
    ----------
    Z : ndarray
        The :math:`(n-1)` by 4 matrix encoding the linkage (hierarchical
        clustering).  See `linkage` documentation for more information on its
        form.
    d : int, optional
        The number of links up to `d` levels below each non-singleton cluster.

    Returns
    -------
    R : ndarray
        A :math:`(n-1)` by 4 matrix where the ``i``'th row contains the link
        statistics for the non-singleton cluster ``i``. The link statistics
        are
        computed over the link heights for links :math:`d` levels below the
        cluster ``i``. ``R[i,0]`` and ``R[i,1]`` are the mean and standard
        deviation of the link heights, respectively; ``R[i,2]`` is the number
        of links included in the calculation; and ``R[i,3]`` is the
        inconsistency coefficient,

        .. math:: \frac{\mathtt{Z[i,2]} - \mathtt{R[i,0]}} {R[i,1]}

    Notes
    -----
    This function behaves similarly to the MATLAB(TM) ``inconsistent``
    function.

    Examples
    --------
    >>> from scipy.cluster.hierarchy import inconsistent, linkage
    >>> from matplotlib import pyplot as plt
    >>> X = [[i] for i in [2, 8, 0, 4, 1, 9, 9, 0]]
    >>> Z = linkage(X, 'ward')
    >>> print(Z)
    [[ 5.          6.          0.          2.        ]
     [ 2.          7.          0.          2.        ]
     [ 0.          4.          1.          2.        ]
     [ 1.          8.          1.15470054  3.        ]
     [ 9.         10.          2.12132034  4.        ]
     [ 3.         12.          4.11096096  5.        ]
     [11.         13.         14.07183949  8.        ]]
    >>> inconsistent(Z)
    array([[ 0.        ,  0.        ,  1.        ,  0.        ],
           [ 0.        ,  0.        ,  1.        ,  0.        ],
           [ 1.        ,  0.        ,  1.        ,  0.        ],
           [ 0.57735027,  0.81649658,  2.        ,  0.70710678],
           [ 1.04044011,  1.06123822,  3.        ,  1.01850858],
           [ 3.11614065,  1.40688837,  2.        ,  0.70710678],
           [ 6.44583366,  6.76770586,  3.        ,  1.12682288]])

    """
    Z = np.asarray(Z, order='c')

    Zs = Z.shape
    is_valid_linkage(Z, throw=True, name='Z')
    if (not d == np.floor(d)) or d < 0:
        raise ValueError('The second argument d must be a nonnegative '
                         'integer value.')

    # Since the C code does not support striding using strides.
    # The dimensions are used instead.
    [Z] = _copy_arrays_if_base_present([Z])

    n = Zs[0] + 1
    R = np.zeros((n - 1, 4), dtype=np.double)

    _hierarchy.inconsistent(Z, R, int(n), int(d))
    return R


def from_mlab_linkage(Z):
    """
    Convert a linkage matrix generated by MATLAB(TM) to a new
    linkage matrix compatible with this module.

    The conversion does two things:

     * the indices are converted from ``1..N`` to ``0..(N-1)`` form,
       and

     * a fourth column ``Z[:,3]`` is added where ``Z[i,3]`` represents the
       number of original observations (leaves) in the non-singleton
       cluster ``i``.

    This function is useful when loading in linkages from legacy data
    files generated by MATLAB.

    Parameters
    ----------
    Z : ndarray
        A linkage matrix generated by MATLAB(TM).

    Returns
    -------
    ZS : ndarray
        A linkage matrix compatible with ``scipy.cluster.hierarchy``.

    """
    Z = np.asarray(Z, dtype=np.double, order='c')
    Zs = Z.shape

    # If it's empty, return it.
    if len(Zs) == 0 or (len(Zs) == 1 and Zs[0] == 0):
        return Z.copy()

    if len(Zs) != 2:
        raise ValueError("The linkage array must be rectangular.")

    # If it contains no rows, return it.
    if Zs[0] == 0:
        return Z.copy()

    Zpart = Z.copy()
    if Zpart[:, 0:2].min() != 1.0 and Zpart[:, 0:2].max() != 2 * Zs[0]:
        raise ValueError('The format of the indices is not 1..N')

    Zpart[:, 0:2] -= 1.0
    CS = np.zeros((Zs[0],), dtype=np.double)
    _hierarchy.calculate_cluster_sizes(Zpart, CS, int(Zs[0]) + 1)
    return np.hstack([Zpart, CS.reshape(Zs[0], 1)])


def to_mlab_linkage(Z):
    """
    Convert a linkage matrix to a MATLAB(TM) compatible one.

    Converts a linkage matrix ``Z`` generated by the linkage function
    of this module to a MATLAB(TM) compatible one. The return linkage
    matrix has the last column removed and the cluster indices are
    converted to ``1..N`` indexing.

    Parameters
    ----------
    Z : ndarray
        A linkage matrix generated by ``scipy.cluster.hierarchy``.

    Returns
    -------
    to_mlab_linkage : ndarray
        A linkage matrix compatible with MATLAB(TM)'s hierarchical
        clustering functions.

        The return linkage matrix has the last column removed
        and the cluster indices are converted to ``1..N`` indexing.

    """
    Z = np.asarray(Z, order='c', dtype=np.double)
    Zs = Z.shape
    if len(Zs) == 0 or (len(Zs) == 1 and Zs[0] == 0):
        return Z.copy()
    is_valid_linkage(Z, throw=True, name='Z')

    ZP = Z[:, 0:3].copy()
    ZP[:, 0:2] += 1.0

    return ZP


def is_monotonic(Z):
    """
    Return True if the linkage passed is monotonic.

    The linkage is monotonic if for every cluster :math:`s` and :math:`t`
    joined, the distance between them is no less than the distance
    between any previously joined clusters.

    Parameters
    ----------
    Z : ndarray
        The linkage matrix to check for monotonicity.

    Returns
    -------
    b : bool
        A boolean indicating whether the linkage is monotonic.

    """
    Z = np.asarray(Z, order='c')
    is_valid_linkage(Z, throw=True, name='Z')

    # We expect the i'th value to be greater than its successor.
    return (Z[1:, 2] >= Z[:-1, 2]).all()


def is_valid_im(R, warning=False, throw=False, name=None):
    """Return True if the inconsistency matrix passed is valid.

    It must be a :math:`n` by 4 array of doubles. The standard
    deviations ``R[:,1]`` must be nonnegative. The link counts
    ``R[:,2]`` must be positive and no greater than :math:`n-1`.

    Parameters
    ----------
    R : ndarray
        The inconsistency matrix to check for validity.
    warning : bool, optional
         When True, issues a Python warning if the linkage
         matrix passed is invalid.
    throw : bool, optional
         When True, throws a Python exception if the linkage
         matrix passed is invalid.
    name : str, optional
         This string refers to the variable name of the invalid
         linkage matrix.

    Returns
    -------
    b : bool
        True if the inconsistency matrix is valid.

    """
    R = np.asarray(R, order='c')
    valid = True
    name_str = "%r " % name if name else ''
    try:
        if type(R) != np.ndarray:
            raise TypeError('Variable %spassed as inconsistency matrix is not
            '
                            'a numpy array.' % name_str)
        if R.dtype != np.double:
            raise TypeError('Inconsistency matrix %smust contain doubles '
                            '(double).' % name_str)
        if len(R.shape) != 2:
            raise ValueError('Inconsistency matrix %smust have shape=2 (i.e. '
                             'be two-dimensional).' % name_str)
        if R.shape[1] != 4:
            raise ValueError('Inconsistency matrix %smust have 4 columns.' %
                             name_str)
        if R.shape[0] < 1:
            raise ValueError('Inconsistency matrix %smust have at least one '
                             'row.' % name_str)
        if (R[:, 0] < 0).any():
            raise ValueError('Inconsistency matrix %scontains negative link '
                             'height means.' % name_str)
        if (R[:, 1] < 0).any():
            raise ValueError('Inconsistency matrix %scontains negative link '
                             'height standard deviations.' % name_str)
        if (R[:, 2] < 0).any():
            raise ValueError('Inconsistency matrix %scontains negative link '
                             'counts.' % name_str)
    except Exception as e:
        if throw:
            raise
        if warning:
            _warning(str(e))
        valid = False

    return valid


def is_valid_linkage(Z, warning=False, throw=False, name=None):
    """
    Check the validity of a linkage matrix.

    A linkage matrix is valid if it is a two dimensional array (type double)
    with :math:`n` rows and 4 columns.  The first two columns must contain
    indices between 0 and :math:`2n-1`. For a given row ``i``, the following
    two expressions have to hold:

    .. math::

        0 \\leq \\mathtt{Z[i,0]} \\leq i+n-1
        0 \\leq Z[i,1] \\leq i+n-1

    I.e. a cluster cannot join another cluster unless the cluster being joined
    has been generated.

    Parameters
    ----------
    Z : array_like
        Linkage matrix.
    warning : bool, optional
        When True, issues a Python warning if the linkage
        matrix passed is invalid.
    throw : bool, optional
        When True, throws a Python exception if the linkage
        matrix passed is invalid.
    name : str, optional
        This string refers to the variable name of the invalid
        linkage matrix.

    Returns
    -------
    b : bool
        True if the inconsistency matrix is valid.

    """
    Z = np.asarray(Z, order='c')
    valid = True
    name_str = "%r " % name if name else ''
    try:
        if type(Z) != np.ndarray:
            raise TypeError('Passed linkage argument %sis not a valid array.'
            %
                            name_str)
        if Z.dtype != np.double:
            raise TypeError('Linkage matrix %smust contain doubles.' %
            name_str)
        if len(Z.shape) != 2:
            raise ValueError('Linkage matrix %smust have shape=2 (i.e. be '
                             'two-dimensional).' % name_str)
        if Z.shape[1] != 4:
            raise ValueError('Linkage matrix %smust have 4 columns.' %
            name_str)
        if Z.shape[0] == 0:
            raise ValueError('Linkage must be computed on at least two '
                             'observations.')
        n = Z.shape[0]
        if n > 1:
            if ((Z[:, 0] < 0).any() or (Z[:, 1] < 0).any()):
                raise ValueError('Linkage %scontains negative indices.' %
                                 name_str)
            if (Z[:, 2] < 0).any():
                raise ValueError('Linkage %scontains negative distances.' %
                                 name_str)
            if (Z[:, 3] < 0).any():
                raise ValueError('Linkage %scontains negative counts.' %
                                 name_str)
        if _check_hierarchy_uses_cluster_before_formed(Z):
            raise ValueError('Linkage %suses non-singleton cluster before '
                             'it is formed.' % name_str)
        if _check_hierarchy_uses_cluster_more_than_once(Z):
            raise ValueError('Linkage %suses the same cluster more than once.'
                             % name_str)
    except Exception as e:
        if throw:
            raise
        if warning:
            _warning(str(e))
        valid = False

    return valid


def _check_hierarchy_uses_cluster_before_formed(Z):
    n = Z.shape[0] + 1
    for i in xrange(0, n - 1):
        if Z[i, 0] >= n + i or Z[i, 1] >= n + i:
            return True
    return False


def _check_hierarchy_uses_cluster_more_than_once(Z):
    n = Z.shape[0] + 1
    chosen = set([])
    for i in xrange(0, n - 1):
        if (Z[i, 0] in chosen) or (Z[i, 1] in chosen) or Z[i, 0] == Z[i, 1]:
            return True
        chosen.add(Z[i, 0])
        chosen.add(Z[i, 1])
    return False


def _check_hierarchy_not_all_clusters_used(Z):
    n = Z.shape[0] + 1
    chosen = set([])
    for i in xrange(0, n - 1):
        chosen.add(int(Z[i, 0]))
        chosen.add(int(Z[i, 1]))
    must_chosen = set(range(0, 2 * n - 2))
    return len(must_chosen.difference(chosen)) > 0


def num_obs_linkage(Z):
    """
    Return the number of original observations of the linkage matrix passed.

    Parameters
    ----------
    Z : ndarray
        The linkage matrix on which to perform the operation.

    Returns
    -------
    n : int
        The number of original observations in the linkage.

    """
    Z = np.asarray(Z, order='c')
    is_valid_linkage(Z, throw=True, name='Z')
    return (Z.shape[0] + 1)


def correspond(Z, Y):
    """
    Check for correspondence between linkage and condensed distance matrices.

    They must have the same number of original observations for
    the check to succeed.

    This function is useful as a sanity check in algorithms that make
    extensive use of linkage and distance matrices that must
    correspond to the same set of original observations.
    Parameters
    ----------
    Z : array_like
        The linkage matrix to check for correspondence.
    Y : array_like
        The condensed distance matrix to check for correspondence.

    Returns
    -------
    b : bool
        A boolean indicating whether the linkage matrix and distance
        matrix could possibly correspond to one another.

    """
    is_valid_linkage(Z, throw=True)
    distance.is_valid_y(Y, throw=True)
    Z = np.asarray(Z, order='c')
    Y = np.asarray(Y, order='c')
    return distance.num_obs_y(Y) == num_obs_linkage(Z)


def fcluster(Z, t, criterion='inconsistent', depth=2, R=None, monocrit=None):
    """
    Form flat clusters from the hierarchical clustering defined by
    the given linkage matrix.

    Parameters
    ----------
    Z : ndarray
        The hierarchical clustering encoded with the matrix returned
        by the `linkage` function.
    t : float
        The threshold to apply when forming flat clusters.
    criterion : str, optional
        The criterion to use in forming flat clusters. This can
        be any of the following values:

          ``inconsistent`` :
              If a cluster node and all its
              descendants have an inconsistent value less than or equal
              to `t` then all its leaf descendants belong to the
              same flat cluster. When no non-singleton cluster meets
              this criterion, every node is assigned to its own
              cluster. (Default)

          ``distance`` :
              Forms flat clusters so that the original
              observations in each flat cluster have no greater a
              cophenetic distance than `t`.

          ``maxclust`` :
              Finds a minimum threshold ``r`` so that
              the cophenetic distance between any two original
              observations in the same flat cluster is no more than
              ``r`` and no more than `t` flat clusters are formed.

          ``monocrit`` :
              Forms a flat cluster from a cluster node c
              with index i when ``monocrit[j] <= t``.

              For example, to threshold on the maximum mean distance
              as computed in the inconsistency matrix R with a
              threshold of 0.8 do::
                  MR = maxRstat(Z, R, 3)
                  cluster(Z, t=0.8, criterion='monocrit', monocrit=MR)

          ``maxclust_monocrit`` :
              Forms a flat cluster from a
              non-singleton cluster node ``c`` when ``monocrit[i] <=
              r`` for all cluster indices ``i`` below and including
              ``c``. ``r`` is minimized such that no more than ``t``
              flat clusters are formed. monocrit must be
              monotonic. For example, to minimize the threshold t on
              maximum inconsistency values so that no more than 3 flat
              clusters are formed, do::

                  MI = maxinconsts(Z, R)
                  cluster(Z, t=3, criterion='maxclust_monocrit', monocrit=MI)

    depth : int, optional
        The maximum depth to perform the inconsistency calculation.
        It has no meaning for the other criteria. Default is 2.
    R : ndarray, optional
        The inconsistency matrix to use for the 'inconsistent'
        criterion. This matrix is computed if not provided.
    monocrit : ndarray, optional
        An array of length n-1. `monocrit[i]` is the
        statistics upon which non-singleton i is thresholded. The
        monocrit vector must be monotonic, i.e. given a node c with
        index i, for all node indices j corresponding to nodes
        below c, ``monocrit[i] >= monocrit[j]``.

    Returns
    -------
    fcluster : ndarray
        An array of length ``n``. ``T[i]`` is the flat cluster number to
        which original observation ``i`` belongs.

    """
    Z = np.asarray(Z, order='c')
    is_valid_linkage(Z, throw=True, name='Z')

    n = Z.shape[0] + 1
    T = np.zeros((n,), dtype='i')

    # Since the C code does not support striding using strides.
    # The dimensions are used instead.
    [Z] = _copy_arrays_if_base_present([Z])

    if criterion == 'inconsistent':
        if R is None:
            R = inconsistent(Z, depth)
        else:
            R = np.asarray(R, order='c')
            is_valid_im(R, throw=True, name='R')
            # Since the C code does not support striding using strides.
            # The dimensions are used instead.
            [R] = _copy_arrays_if_base_present([R])
        _hierarchy.cluster_in(Z, R, T, float(t), int(n))
    elif criterion == 'distance':
        _hierarchy.cluster_dist(Z, T, float(t), int(n))
    elif criterion == 'maxclust':
        _hierarchy.cluster_maxclust_dist(Z, T, int(n), int(t))
    elif criterion == 'monocrit':
        [monocrit] = _copy_arrays_if_base_present([monocrit])
        _hierarchy.cluster_monocrit(Z, monocrit, T, float(t), int(n))
    elif criterion == 'maxclust_monocrit':
        [monocrit] = _copy_arrays_if_base_present([monocrit])
        _hierarchy.cluster_maxclust_monocrit(Z, monocrit, T, int(n), int(t))
    else:
        raise ValueError('Invalid cluster formation criterion: %s'
                         % str(criterion))
    return T


def fclusterdata(X, t, criterion='inconsistent',
                 metric='euclidean', depth=2, method='single', R=None):
    """
    Cluster observation data using a given metric.

    Clusters the original observations in the n-by-m data
    matrix X (n observations in m dimensions), using the euclidean
    distance metric to calculate distances between original observations,
    performs hierarchical clustering using the single linkage algorithm,
    and forms flat clusters using the inconsistency method with `t` as the
    cut-off threshold.

    A one-dimensional array ``T`` of length ``n`` is returned. ``T[i]`` is
    the index of the flat cluster to which the original observation ``i``
    belongs.

    Parameters
    ----------
    X : (N, M) ndarray
        N by M data matrix with N observations in M dimensions.
    t : float
        The threshold to apply when forming flat clusters.
    criterion : str, optional
        Specifies the criterion for forming flat clusters.  Valid
        values are 'inconsistent' (default), 'distance', or 'maxclust'
        cluster formation algorithms. See `fcluster` for descriptions.
    metric : str, optional
        The distance metric for calculating pairwise distances. See
        ``distance.pdist`` for descriptions and linkage to verify
        compatibility with the linkage method.
    depth : int, optional
        The maximum depth for the inconsistency calculation. See
        `inconsistent` for more information.
    method : str, optional
        The linkage method to use (single, complete, average,
        weighted, median centroid, ward). See `linkage` for more
        information. Default is "single".
    R : ndarray, optional
        The inconsistency matrix. It will be computed if necessary
        if it is not passed.

    Returns
    -------
    fclusterdata : ndarray
        A vector of length n. T[i] is the flat cluster number to
        which original observation i belongs.

    See Also
    --------
    scipy.spatial.distance.pdist : pairwise distance metrics

    Notes
    -----
    This function is similar to the MATLAB function ``clusterdata``.
    """
    X = np.asarray(X, order='c', dtype=np.double)

    if type(X) != np.ndarray or len(X.shape) != 2:
        raise TypeError('The observation matrix X must be an n by m numpy '
                        'array.')

    Y = distance.pdist(X, metric=metric)
    Z = linkage(Y, method=method)
    if R is None:
        R = inconsistent(Z, d=depth)
    else:
        R = np.asarray(R, order='c')
    T = fcluster(Z, criterion=criterion, depth=depth, R=R, t=t)
    return T


def leaves_list(Z):
    """
    Return a list of leaf node ids.

    The return corresponds to the observation vector index as it appears
    in the tree from left to right. Z is a linkage matrix.

    Parameters
    ----------
    Z : ndarray
        The hierarchical clustering encoded as a matrix.  `Z` is
        a linkage matrix.  See `linkage` for more information.

    Returns
    -------
    leaves_list : ndarray
        The list of leaf node ids.

    """
    Z = np.asarray(Z, order='c')
    is_valid_linkage(Z, throw=True, name='Z')
    n = Z.shape[0] + 1
    ML = np.zeros((n,), dtype='i')
    [Z] = _copy_arrays_if_base_present([Z])
    _hierarchy.prelist(Z, ML, int(n))
    return ML


# Maps number of leaves to text size.
#
# p <= 20, size="12"
# 20 < p <= 30, size="10"
# 30 < p <= 50, size="8"
# 50 < p <= np.inf, size="6"

_dtextsizes = {20: 12, 30: 10, 50: 8, 85: 6, np.inf: 5}
_drotation = {20: 0, 40: 45, np.inf: 90}
_dtextsortedkeys = list(_dtextsizes.keys())
_dtextsortedkeys.sort()
_drotationsortedkeys = list(_drotation.keys())
_drotationsortedkeys.sort()


def _remove_dups(L):
    """
    Remove duplicates AND preserve the original order of the elements.
    The set class is not guaranteed to do this.
    """
    seen_before = set([])
    L2 = []
    for i in L:
        if i not in seen_before:
            seen_before.add(i)
            L2.append(i)
    return L2


def _get_tick_text_size(p):
    for k in _dtextsortedkeys:
        if p <= k:
            return _dtextsizes[k]


def _get_tick_rotation(p):
    for k in _drotationsortedkeys:
        if p <= k:
            return _drotation[k]


def _plot_dendrogram(icoords, dcoords, ivl, p, n, mh, orientation,
                     no_labels, color_list, leaf_font_size=None,
                     leaf_rotation=None, contraction_marks=None,
                     ax=None, above_threshold_color='b'):
    # Import matplotlib here so that it's not imported unless dendrograms
    # are plotted. Raise an informative error if importing fails.
    try:
        # if an axis is provided, don't use pylab at all
        if ax is None:
            import matplotlib.pylab
        import matplotlib.patches
        import matplotlib.collections
    except ImportError:
        raise ImportError("You must install the matplotlib library to plot "
                          "the dendrogram. Use no_plot=True to calculate the "
                          "dendrogram without plotting.")

    if ax is None:
        ax = matplotlib.pylab.gca()
        # if we're using pylab, we want to trigger a draw at the end
        trigger_redraw = True
    else:
        trigger_redraw = False

    # Independent variable plot width
    ivw = len(ivl) * 10
    # Dependent variable plot height
    dvw = mh + mh * 0.05

    iv_ticks = np.arange(5, len(ivl) * 10 + 5, 10)
    if orientation in ('top', 'bottom'):
        if orientation == 'top':
            ax.set_ylim([0, dvw])
            ax.set_xlim([0, ivw])
        else:
            ax.set_ylim([dvw, 0])
            ax.set_xlim([0, ivw])

        xlines = icoords
        ylines = dcoords
        if no_labels:
            ax.set_xticks([])
            ax.set_xticklabels([])
        else:
            ax.set_xticks(iv_ticks)

            if orientation == 'top':
                ax.xaxis.set_ticks_position('bottom')
            else:
                ax.xaxis.set_ticks_position('top')

            # Make the tick marks invisible because they cover up the links
            for line in ax.get_xticklines():
                line.set_visible(False)

            leaf_rot = (float(_get_tick_rotation(len(ivl)))
                        if (leaf_rotation is None) else leaf_rotation)
            leaf_font = (float(_get_tick_text_size(len(ivl)))
                         if (leaf_font_size is None) else leaf_font_size)
            ax.set_xticklabels(ivl, rotation=leaf_rot, size=leaf_font)

    elif orientation in ('left', 'right'):
        if orientation == 'left':
            ax.set_xlim([dvw, 0])
            ax.set_ylim([0, ivw])
        else:
            ax.set_xlim([0, dvw])
            ax.set_ylim([0, ivw])

        xlines = dcoords
        ylines = icoords
        if no_labels:
            ax.set_yticks([])
            ax.set_yticklabels([])
        else:
            ax.set_yticks(iv_ticks)

            if orientation == 'left':
                ax.yaxis.set_ticks_position('right')
            else:
                ax.yaxis.set_ticks_position('left')

            # Make the tick marks invisible because they cover up the links
            for line in ax.get_yticklines():
                line.set_visible(False)

            leaf_font = (float(_get_tick_text_size(len(ivl)))
                         if (leaf_font_size is None) else leaf_font_size)

            if leaf_rotation is not None:
                ax.set_yticklabels(ivl, rotation=leaf_rotation,
                size=leaf_font)
            else:
                ax.set_yticklabels(ivl, size=leaf_font)

    # Let's use collections instead. This way there is a separate legend item
    # for each tree grouping, rather than stupidly one for each line segment.
    colors_used = _remove_dups(color_list)
    color_to_lines = {}
    for color in colors_used:
        color_to_lines[color] = []
    for (xline, yline, color) in zip(xlines, ylines, color_list):
        color_to_lines[color].append(list(zip(xline, yline)))

    colors_to_collections = {}
    # Construct the collections.
    for color in colors_used:
        coll = matplotlib.collections.LineCollection(color_to_lines[color],
                                                     colors=(color,))
        colors_to_collections[color] = coll

    # Add all the groupings below the color threshold.
    for color in colors_used:
        if color != above_threshold_color:
            ax.add_collection(colors_to_collections[color])
    # If there's a grouping of links above the color threshold, it goes last.
    if above_threshold_color in colors_to_collections:
        ax.add_collection(colors_to_collections[above_threshold_color])

    if contraction_marks is not None:
        Ellipse = matplotlib.patches.Ellipse
        for (x, y) in contraction_marks:
            if orientation in ('left', 'right'):
                e = Ellipse((y, x), width=dvw / 100, height=1.0)
            else:
                e = Ellipse((x, y), width=1.0, height=dvw / 100)
            ax.add_artist(e)
            e.set_clip_box(ax.bbox)
            e.set_alpha(0.5)
            e.set_facecolor('k')

    if trigger_redraw:
        matplotlib.pylab.draw_if_interactive()


_link_line_colors = ['g', 'r', 'c', 'm', 'y', 'k']


def set_link_color_palette(palette):
    """
    Set list of matplotlib color codes for use by dendrogram.

    Note that this palette is global (i.e. setting it once changes the colors
    for all subsequent calls to `dendrogram`) and that it affects only the
    the colors below ``color_threshold``.

    Note that `dendrogram` also accepts a custom coloring function through its
    ``link_color_func`` keyword, which is more flexible and non-global.

    Parameters
    ----------
    palette : list of str or None
        A list of matplotlib color codes.  The order of the color codes is the
        order in which the colors are cycled through when color thresholding
        in
        the dendrogram.

        If ``None``, resets the palette to its default (which is
        ``['g', 'r', 'c', 'm', 'y', 'k']``).

    Returns
    -------
    None

    See Also
    --------
    dendrogram

    Notes
    -----
    Ability to reset the palette with ``None`` added in Scipy 0.17.0.

    Examples
    --------
    >>> from scipy.cluster import hierarchy
    >>> ytdist = np.array([662., 877., 255., 412., 996., 295., 468., 268.,
    ...                    400., 754., 564., 138., 219., 869., 669.])
    >>> Z = hierarchy.linkage(ytdist, 'single')
    >>> dn = hierarchy.dendrogram(Z, no_plot=True)
    >>> dn['color_list']
    ['g', 'b', 'b', 'b', 'b']
    >>> hierarchy.set_link_color_palette(['c', 'm', 'y', 'k'])
    >>> dn = hierarchy.dendrogram(Z, no_plot=True)
    >>> dn['color_list']
    ['c', 'b', 'b', 'b', 'b']
    >>> dn = hierarchy.dendrogram(Z, no_plot=True, color_threshold=267,
    ...                           above_threshold_color='k')
    >>> dn['color_list']
    ['c', 'm', 'm', 'k', 'k']

    Now reset the color palette to its default:

    >>> hierarchy.set_link_color_palette(None)

    """
    if palette is None:
        # reset to its default
        palette = ['g', 'r', 'c', 'm', 'y', 'k']
    elif type(palette) not in (list, tuple):
        raise TypeError("palette must be a list or tuple")
    _ptypes = [isinstance(p, string_types) for p in palette]

    if False in _ptypes:
        raise TypeError("all palette list elements must be color strings")

    for i in list(_link_line_colors):
        _link_line_colors.remove(i)
    _link_line_colors.extend(list(palette))


def dendrogram(Z, p=30, truncate_mode=None, color_threshold=None,
               get_leaves=True, orientation='top', labels=None,
               count_sort=False, distance_sort=False, show_leaf_counts=True,
               no_plot=False, no_labels=False, leaf_font_size=None,
               leaf_rotation=None, leaf_label_func=None,
               show_contracted=False, link_color_func=None, ax=None,
               above_threshold_color='b'):
    """
    Plot the hierarchical clustering as a dendrogram.

    The dendrogram illustrates how each cluster is
    composed by drawing a U-shaped link between a non-singleton
    cluster and its children.  The top of the U-link indicates a
    cluster merge.  The two legs of the U-link indicate which clusters
    were merged.  The length of the two legs of the U-link represents
    the distance between the child clusters.  It is also the
    cophenetic distance between original observations in the two
    children clusters.

    Parameters
    ----------
    Z : ndarray
        The linkage matrix encoding the hierarchical clustering to
        render as a dendrogram. See the ``linkage`` function for more
        information on the format of ``Z``.
    p : int, optional
        The ``p`` parameter for ``truncate_mode``.
    truncate_mode : str, optional
        The dendrogram can be hard to read when the original
        observation matrix from which the linkage is derived is
        large. Truncation is used to condense the dendrogram. There
        are several modes:

        ``None``
          No truncation is performed (default).
          Note: ``'none'`` is an alias for ``None`` that's kept for
          backward compatibility.

        ``'lastp'``
          The last ``p`` non-singleton clusters formed in the linkage are the
          only non-leaf nodes in the linkage; they correspond to rows
          ``Z[n-p-2:end]`` in ``Z``. All other non-singleton clusters are
          contracted into leaf nodes.

        ``'level'``
          No more than ``p`` levels of the dendrogram tree are displayed.
          A "level" includes all nodes with ``p`` merges from the last merge.

          Note: ``'mtica'`` is an alias for ``'level'`` that's kept for
          backward compatibility.

    color_threshold : double, optional
        For brevity, let :math:`t` be the ``color_threshold``.
        Colors all the descendent links below a cluster node
        :math:`k` the same color if :math:`k` is the first node below
        the cut threshold :math:`t`. All links connecting nodes with
        distances greater than or equal to the threshold are colored
        blue. If :math:`t` is less than or equal to zero, all nodes
        are colored blue. If ``color_threshold`` is None or
        'default', corresponding with MATLAB(TM) behavior, the
        threshold is set to ``0.7*max(Z[:,2])``.
    get_leaves : bool, optional
        Includes a list ``R['leaves']=H`` in the result
        dictionary. For each :math:`i`, ``H[i] == j``, cluster node
        ``j`` appears in position ``i`` in the left-to-right traversal
        of the leaves, where :math:`j < 2n-1` and :math:`i < n`.
    orientation : str, optional
        The direction to plot the dendrogram, which can be any
        of the following strings:

        ``'top'``
          Plots the root at the top, and plot descendent links going
          downwards.
          (default).

        ``'bottom'``
          Plots the root at the bottom, and plot descendent links going
          upwards.

        ``'left'``
          Plots the root at the left, and plot descendent links going right.

        ``'right'``
          Plots the root at the right, and plot descendent links going left.

    labels : ndarray, optional
        By default ``labels`` is None so the index of the original observation
        is used to label the leaf nodes.  Otherwise, this is an :math:`n`
        -sized list (or tuple). The ``labels[i]`` value is the text to put
        under the :math:`i` th leaf node only if it corresponds to an original
        observation and not a non-singleton cluster.
    count_sort : str or bool, optional
        For each node n, the order (visually, from left-to-right) n's
        two descendent links are plotted is determined by this
        parameter, which can be any of the following values:

        ``False``
          Nothing is done.

        ``'ascending'`` or ``True``
          The child with the minimum number of original objects in its cluster
          is plotted first.

        ``'descendent'``
          The child with the maximum number of original objects in its cluster
          is plotted first.

        Note ``distance_sort`` and ``count_sort`` cannot both be True.
    distance_sort : str or bool, optional
        For each node n, the order (visually, from left-to-right) n's
        two descendent links are plotted is determined by this
        parameter, which can be any of the following values:

        ``False``
          Nothing is done.

        ``'ascending'`` or ``True``
          The child with the minimum distance between its direct descendents
          is
          plotted first.

        ``'descending'``
          The child with the maximum distance between its direct descendents
          is
          plotted first.

        Note ``distance_sort`` and ``count_sort`` cannot both be True.
    show_leaf_counts : bool, optional
         When True, leaf nodes representing :math:`k>1` original
         observation are labeled with the number of observations they
         contain in parentheses.
    no_plot : bool, optional
        When True, the final rendering is not performed. This is
        useful if only the data structures computed for the rendering
        are needed or if matplotlib is not available.
    no_labels : bool, optional
        When True, no labels appear next to the leaf nodes in the
        rendering of the dendrogram.
    leaf_rotation : double, optional
        Specifies the angle (in degrees) to rotate the leaf
        labels. When unspecified, the rotation is based on the number of
        nodes in the dendrogram (default is 0).
    leaf_font_size : int, optional
        Specifies the font size (in points) of the leaf labels. When
        unspecified, the size based on the number of nodes in the
        dendrogram.
    leaf_label_func : lambda or function, optional
        When leaf_label_func is a callable function, for each
        leaf with cluster index :math:`k < 2n-1`. The function
        is expected to return a string with the label for the
        leaf.
        Indices :math:`k < n` correspond to original observations
        while indices :math:`k \\geq n` correspond to non-singleton
        clusters.

        For example, to label singletons with their node id and
        non-singletons with their id, count, and inconsistency
        coefficient, simply do::

            # First define the leaf label function.
            def llf(id):
                if id < n:
                    return str(id)
                else:
                    return '[%d %d %1.2f]' % (id, count, R[n-id,3])
            # The text for the leaf nodes is going to be big so force
            # a rotation of 90 degrees.
            dendrogram(Z, leaf_label_func=llf, leaf_rotation=90)

    show_contracted : bool, optional
        When True the heights of non-singleton nodes contracted
        into a leaf node are plotted as crosses along the link
        connecting that leaf node.  This really is only useful when
        truncation is used (see ``truncate_mode`` parameter).
    link_color_func : callable, optional
        If given, `link_color_function` is called with each non-singleton id
        corresponding to each U-shaped link it will paint. The function is
        expected to return the color to paint the link, encoded as a
        matplotlib
        color string code. For example::

            dendrogram(Z, link_color_func=lambda k: colors[k])

        colors the direct links below each untruncated non-singleton node
        ``k`` using ``colors[k]``.
    ax : matplotlib Axes instance, optional
        If None and `no_plot` is not True, the dendrogram will be plotted
        on the current axes.  Otherwise if `no_plot` is not True the
        dendrogram will be plotted on the given ``Axes`` instance. This can be
        useful if the dendrogram is part of a more complex figure.
    above_threshold_color : str, optional
        This matplotlib color string sets the color of the links above the
        color_threshold. The default is 'b'.

    Returns
    -------
    R : dict
        A dictionary of data structures computed to render the
        dendrogram. Its has the following keys:

        ``'color_list'``
          A list of color names. The k'th element represents the color of the
          k'th link.

        ``'icoord'`` and ``'dcoord'``
          Each of them is a list of lists. Let ``icoord = [I1, I2, ..., Ip]``
          where ``Ik = [xk1, xk2, xk3, xk4]`` and ``dcoord = [D1, D2, ...,
          Dp]``
          where ``Dk = [yk1, yk2, yk3, yk4]``, then the k'th link painted is
          ``(xk1, yk1)`` - ``(xk2, yk2)`` - ``(xk3, yk3)`` - ``(xk4, yk4)``.

        ``'ivl'``
          A list of labels corresponding to the leaf nodes.
        ``'leaves'``
          For each i, ``H[i] == j``, cluster node ``j`` appears in position
          ``i`` in the left-to-right traversal of the leaves, where
          :math:`j < 2n-1` and :math:`i < n`. If ``j`` is less than ``n``, the
          ``i``-th leaf node corresponds to an original observation.
          Otherwise, it corresponds to a non-singleton cluster.

    See Also
    --------
    linkage, set_link_color_palette

    Notes
    -----
    It is expected that the distances in ``Z[:,2]`` be monotonic, otherwise
    crossings appear in the dendrogram.

    Examples
    --------
    >>> from scipy.cluster import hierarchy
    >>> import matplotlib.pyplot as plt

    A very basic example:

    >>> ytdist = np.array([662., 877., 255., 412., 996., 295., 468., 268.,
    ...                    400., 754., 564., 138., 219., 869., 669.])
    >>> Z = hierarchy.linkage(ytdist, 'single')
    >>> plt.figure()
    >>> dn = hierarchy.dendrogram(Z)

    Now plot in given axes, improve the color scheme and use both vertical and
    horizontal orientations:

    >>> hierarchy.set_link_color_palette(['m', 'c', 'y', 'k'])
    >>> fig, axes = plt.subplots(1, 2, figsize=(8, 3))
    >>> dn1 = hierarchy.dendrogram(Z, ax=axes[0], above_threshold_color='y',
    ...                            orientation='top')
    >>> dn2 = hierarchy.dendrogram(Z, ax=axes[1],
    ...                            above_threshold_color='#bcbddc',
    ...                            orientation='right')
    >>> hierarchy.set_link_color_palette(None)  # reset to default after use
    >>> plt.show()

    """
    # This feature was thought about but never implemented (still useful?):
    #
    #         ... = dendrogram(..., leaves_order=None)
    #
    #         Plots the leaves in the order specified by a vector of
    #         original observation indices. If the vector contains duplicates
    #         or results in a crossing, an exception will be thrown. Passing
    #         None orders leaf nodes based on the order they appear in the
    #         pre-order traversal.
    Z = np.asarray(Z, order='c')

    if orientation not in ["top", "left", "bottom", "right"]:
        raise ValueError("orientation must be one of 'top', 'left', "
                         "'bottom', or 'right'")

    is_valid_linkage(Z, throw=True, name='Z')
    Zs = Z.shape
    n = Zs[0] + 1
    if type(p) in (int, float):
        p = int(p)
    else:
        raise TypeError('The second argument must be a number')

    if truncate_mode not in ('lastp', 'mlab', 'mtica', 'level', 'none', None):
        # 'mlab' and 'mtica' are kept working for backwards compat.
        raise ValueError('Invalid truncation mode.')

    if truncate_mode == 'lastp' or truncate_mode == 'mlab':
        if p > n or p == 0:
            p = n

    if truncate_mode == 'mtica':
        # 'mtica' is an alias
        truncate_mode = 'level'

    if truncate_mode == 'level':
        if p <= 0:
            p = np.inf

    if get_leaves:
        lvs = []
    else:
        lvs = None

    icoord_list = []
    dcoord_list = []
    color_list = []
    current_color = [0]
    currently_below_threshold = [False]
    ivl = []  # list of leaves

    if color_threshold is None or (isinstance(color_threshold, string_types)
    and
                                   color_threshold == 'default'):
        color_threshold = max(Z[:, 2]) * 0.7

    R = {'icoord': icoord_list, 'dcoord': dcoord_list, 'ivl': ivl,
         'leaves': lvs, 'color_list': color_list}

    # Empty list will be filled in _dendrogram_calculate_info
    contraction_marks = [] if show_contracted else None

    _dendrogram_calculate_info(
        Z=Z, p=p,
        truncate_mode=truncate_mode,
        color_threshold=color_threshold,
        get_leaves=get_leaves,
        orientation=orientation,
        labels=labels,
        count_sort=count_sort,
        distance_sort=distance_sort,
        show_leaf_counts=show_leaf_counts,
        i=2*n - 2,
        iv=0.0,
        ivl=ivl,
        n=n,
        icoord_list=icoord_list,
        dcoord_list=dcoord_list,
        lvs=lvs,
        current_color=current_color,
        color_list=color_list,
        currently_below_threshold=currently_below_threshold,
        leaf_label_func=leaf_label_func,
        contraction_marks=contraction_marks,
        link_color_func=link_color_func,
        above_threshold_color=above_threshold_color)

    if not no_plot:
        mh = max(Z[:, 2])
        _plot_dendrogram(icoord_list, dcoord_list, ivl, p, n, mh, orientation,
                         no_labels, color_list,
                         leaf_font_size=leaf_font_size,
                         leaf_rotation=leaf_rotation,
                         contraction_marks=contraction_marks,
                         ax=ax,
                         above_threshold_color=above_threshold_color)

    return R


def _append_singleton_leaf_node(Z, p, n, level, lvs, ivl, leaf_label_func,
                                i, labels):
    # If the leaf id structure is not None and is a list then the caller
    # to dendrogram has indicated that cluster id's corresponding to the
    # leaf nodes should be recorded.

    if lvs is not None:
        lvs.append(int(i))

    # If leaf node labels are to be displayed...
    if ivl is not None:
        # If a leaf_label_func has been provided, the label comes from the
        # string returned from the leaf_label_func, which is a function
        # passed to dendrogram.
        if leaf_label_func:
            ivl.append(leaf_label_func(int(i)))
        else:
            # Otherwise, if the dendrogram caller has passed a labels list
            # for the leaf nodes, use it.
            if labels is not None:
                ivl.append(labels[int(i - n)])
            else:
                # Otherwise, use the id as the label for the leaf.x
                ivl.append(str(int(i)))


def _append_nonsingleton_leaf_node(Z, p, n, level, lvs, ivl, leaf_label_func,
                                   i, labels, show_leaf_counts):
    # If the leaf id structure is not None and is a list then the caller
    # to dendrogram has indicated that cluster id's corresponding to the
    # leaf nodes should be recorded.

    if lvs is not None:
        lvs.append(int(i))
    if ivl is not None:
        if leaf_label_func:
            ivl.append(leaf_label_func(int(i)))
        else:
            if show_leaf_counts:
                ivl.append("(" + str(int(Z[i - n, 3])) + ")")
            else:
                ivl.append("")


def _append_contraction_marks(Z, iv, i, n, contraction_marks):
    _append_contraction_marks_sub(Z, iv, int(Z[i - n, 0]), n,
    contraction_marks)
    _append_contraction_marks_sub(Z, iv, int(Z[i - n, 1]), n,
    contraction_marks)
```

```
def _append_contraction_marks_sub(Z, iv, i, n, contraction_marks):
    if i >= n:
        contraction_marks.append((iv, Z[i - n, 2]))
        _append_contraction_marks_sub(Z, iv, int(Z[i - n, 0]), n,
        contraction_marks)
        _append_contraction_marks_sub(Z, iv, int(Z[i - n, 1]), n,
        contraction_marks)


def _dendrogram_calculate_info(Z, p, truncate_mode,
                               color_threshold=np.inf, get_leaves=True,
                               orientation='top', labels=None,
                               count_sort=False, distance_sort=False,
                               show_leaf_counts=False, i=-1, iv=0.0,
                               ivl=[], n=0, icoord_list=[], dcoord_list=[],
                               lvs=None, mhr=False,
                               current_color=[], color_list=[],
                               currently_below_threshold=[],
                               leaf_label_func=None, level=0,
                               contraction_marks=None,
                               link_color_func=None,
                               above_threshold_color='b'):
    """
    Calculate the endpoints of the links as well as the labels for the
    the dendrogram rooted at the node with index i. iv is the independent
    variable value to plot the left-most leaf node below the root node i
    (if orientation='top', this would be the left-most x value where the
    plotting of this root node i and its descendents should begin).

    ivl is a list to store the labels of the leaf nodes. The leaf_label_func
    is called whenever ivl != None, labels == None, and
    leaf_label_func != None. When ivl != None and labels != None, the
    labels list is used only for labeling the leaf nodes. When
    ivl == None, no labels are generated for leaf nodes.

    When get_leaves==True, a list of leaves is built as they are visited
    in the dendrogram.

    Returns a tuple with l being the independent variable coordinate that
    corresponds to the midpoint of cluster to the left of cluster i if
    i is non-singleton, otherwise the independent coordinate of the leaf
    node if i is a leaf node.

    Returns
    -------
    A tuple (left, w, h, md), where:

      * left is the independent variable coordinate of the center of the
        the U of the subtree

      * w is the amount of space used for the subtree (in independent
        variable units)

      * h is the height of the subtree in dependent variable units

      * md is the ``max(Z[*,2]``) for all nodes ``*`` below and including
        the target node.

    """
    if n == 0:
        raise ValueError("Invalid singleton cluster count n.")
    if i == -1:
        raise ValueError("Invalid root cluster index i.")

    if truncate_mode == 'lastp':
        # If the node is a leaf node but corresponds to a non-singleton
        # cluster, its label is either the empty string or the number of
        # original observations belonging to cluster i.
        if 2*n - p > i >= n:
            d = Z[i - n, 2]
            _append_nonsingleton_leaf_node(Z, p, n, level, lvs, ivl,
                                           leaf_label_func, i, labels,
                                           show_leaf_counts)
            if contraction_marks is not None:
                _append_contraction_marks(Z, iv + 5.0, i, n,
                contraction_marks)
            return (iv + 5.0, 10.0, 0.0, d)
        elif i < n:
            _append_singleton_leaf_node(Z, p, n, level, lvs, ivl,
                                        leaf_label_func, i, labels)
            return (iv + 5.0, 10.0, 0.0, 0.0)
    elif truncate_mode == 'level':
        if i > n and level > p:
            d = Z[i - n, 2]
            _append_nonsingleton_leaf_node(Z, p, n, level, lvs, ivl,
                                           leaf_label_func, i, labels,
                                           show_leaf_counts)
            if contraction_marks is not None:
                _append_contraction_marks(Z, iv + 5.0, i, n,
                contraction_marks)
            return (iv + 5.0, 10.0, 0.0, d)
        elif i < n:
            _append_singleton_leaf_node(Z, p, n, level, lvs, ivl,
                                        leaf_label_func, i, labels)
            return (iv + 5.0, 10.0, 0.0, 0.0)
    elif truncate_mode in ('mlab',):
        msg = "Mode 'mlab' is deprecated in scipy 0.19.0 (it never worked)."
        warnings.warn(msg, DeprecationWarning)

    # Otherwise, only truncate if we have a leaf node.
    #
    # Only place leaves if they correspond to original observations.
    if i < n:
        _append_singleton_leaf_node(Z, p, n, level, lvs, ivl,
                                    leaf_label_func, i, labels)
        return (iv + 5.0, 10.0, 0.0, 0.0)

    # !!! Otherwise, we don't have a leaf node, so work on plotting a
    # non-leaf node.
    # Actual indices of a and b
    aa = int(Z[i - n, 0])
    ab = int(Z[i - n, 1])
    if aa > n:
        # The number of singletons below cluster a
        na = Z[aa - n, 3]
        # The distance between a's two direct children.
        da = Z[aa - n, 2]
    else:
        na = 1
        da = 0.0
    if ab > n:
        nb = Z[ab - n, 3]
        db = Z[ab - n, 2]
    else:
        nb = 1
        db = 0.0

    if count_sort == 'ascending' or count_sort:
        # If a has a count greater than b, it and its descendents should
        # be drawn to the right. Otherwise, to the left.
        if na > nb:
            # The cluster index to draw to the left (ua) will be ab
            # and the one to draw to the right (ub) will be aa
            ua = ab
            ub = aa
        else:
            ua = aa
            ub = ab
    elif count_sort == 'descending':
        # If a has a count less than or equal to b, it and its
        # descendents should be drawn to the left. Otherwise, to
        # the right.
        if na > nb:
            ua = aa
            ub = ab
        else:
            ua = ab
            ub = aa
    elif distance_sort == 'ascending' or distance_sort:
        # If a has a distance greater than b, it and its descendents should
        # be drawn to the right. Otherwise, to the left.
        if da > db:
            ua = ab
            ub = aa
        else:
            ua = aa
            ub = ab
    elif distance_sort == 'descending':
        # If a has a distance less than or equal to b, it and its
        # descendents should be drawn to the left. Otherwise, to
        # the right.
        if da > db:
            ua = aa
            ub = ab
        else:
            ua = ab
            ub = aa
    else:
        ua = aa
        ub = ab

    # Updated iv variable and the amount of space used.
    (uiva, uwa, uah, uamd) = \
        _dendrogram_calculate_info(
            Z=Z, p=p,
            truncate_mode=truncate_mode,
            color_threshold=color_threshold,
            get_leaves=get_leaves,
            orientation=orientation,
            labels=labels,
            count_sort=count_sort,
            distance_sort=distance_sort,
            show_leaf_counts=show_leaf_counts,
            i=ua, iv=iv, ivl=ivl, n=n,
            icoord_list=icoord_list,
            dcoord_list=dcoord_list, lvs=lvs,
            current_color=current_color,
            color_list=color_list,
            currently_below_threshold=currently_below_threshold,
            leaf_label_func=leaf_label_func,
            level=level + 1, contraction_marks=contraction_marks,
            link_color_func=link_color_func,
            above_threshold_color=above_threshold_color)

    h = Z[i - n, 2]
    if h >= color_threshold or color_threshold <= 0:
        c = above_threshold_color

        if currently_below_threshold[0]:
            current_color[0] = (current_color[0] + 1) % len(_link_line_colors)
        currently_below_threshold[0] = False
    else:
        currently_below_threshold[0] = True
        c = _link_line_colors[current_color[0]]

    (uivb, uwb, ubh, ubmd) = \
        _dendrogram_calculate_info(
            Z=Z, p=p,
            truncate_mode=truncate_mode,
            color_threshold=color_threshold,
            get_leaves=get_leaves,
            orientation=orientation,
            labels=labels,
            count_sort=count_sort,
            distance_sort=distance_sort,
            show_leaf_counts=show_leaf_counts,
            i=ub, iv=iv + uwa, ivl=ivl, n=n,
            icoord_list=icoord_list,
            dcoord_list=dcoord_list, lvs=lvs,
            current_color=current_color,
            color_list=color_list,
            currently_below_threshold=currently_below_threshold,
            leaf_label_func=leaf_label_func,
            level=level + 1, contraction_marks=contraction_marks,
            link_color_func=link_color_func,
            above_threshold_color=above_threshold_color)

    max_dist = max(uamd, ubmd, h)

    icoord_list.append([uiva, uiva, uivb, uivb])
    dcoord_list.append([uah, h, h, ubh])
    if link_color_func is not None:
        v = link_color_func(int(i))
        if not isinstance(v, string_types):
            raise TypeError("link_color_func must return a matplotlib "
                            "color string!")
        color_list.append(v)
    else:
        color_list.append(c)

    return (((uiva + uivb) / 2), uwa + uwb, h, max_dist)


def is_isomorphic(T1, T2):
    """
    Determine if two different cluster assignments are equivalent.

    Parameters
    ----------
    T1 : array_like
        An assignment of singleton cluster ids to flat cluster ids.
    T2 : array_like
        An assignment of singleton cluster ids to flat cluster ids.
    Returns
    -------
    b : bool
        Whether the flat cluster assignments `T1` and `T2` are
        equivalent.

    """
    T1 = np.asarray(T1, order='c')
    T2 = np.asarray(T2, order='c')

    if type(T1) != np.ndarray:
        raise TypeError('T1 must be a numpy array.')
    if type(T2) != np.ndarray:
        raise TypeError('T2 must be a numpy array.')

    T1S = T1.shape
    T2S = T2.shape

    if len(T1S) != 1:
        raise ValueError('T1 must be one-dimensional.')
    if len(T2S) != 1:
        raise ValueError('T2 must be one-dimensional.')
    if T1S[0] != T2S[0]:
        raise ValueError('T1 and T2 must have the same number of elements.')
    n = T1S[0]
    d1 = {}
    d2 = {}
    for i in xrange(0, n):
        if T1[i] in d1:
            if not T2[i] in d2:
                return False
            if d1[T1[i]] != T2[i] or d2[T2[i]] != T1[i]:
                return False
        elif T2[i] in d2:
            return False
        else:
            d1[T1[i]] = T2[i]
            d2[T2[i]] = T1[i]
    return True


def maxdists(Z):
    """
    Return the maximum distance between any non-singleton cluster.

    Parameters
    ----------
    Z : ndarray
        The hierarchical clustering encoded as a matrix. See
        ``linkage`` for more information.

    Returns
    -------
    maxdists : ndarray
        A ``(n-1)`` sized numpy array of doubles; ``MD[i]`` represents
        the maximum distance between any cluster (including
        singletons) below and including the node with index i. More
        specifically, ``MD[i] = Z[Q(i)-n, 2].max()`` where ``Q(i)`` is the
        set of all node indices below and including node i.

    """
    Z = np.asarray(Z, order='c', dtype=np.double)
    is_valid_linkage(Z, throw=True, name='Z')
    n = Z.shape[0] + 1
    MD = np.zeros((n - 1,))
    [Z] = _copy_arrays_if_base_present([Z])
    _hierarchy.get_max_dist_for_each_cluster(Z, MD, int(n))
    return MD


def maxinconsts(Z, R):
    """
    Return the maximum inconsistency coefficient for each
    non-singleton cluster and its descendents.

    Parameters
    ----------
    Z : ndarray
        The hierarchical clustering encoded as a matrix. See
        `linkage` for more information.
    R : ndarray
        The inconsistency matrix.

    Returns
    -------
    MI : ndarray
        A monotonic ``(n-1)``-sized numpy array of doubles.

    """
    Z = np.asarray(Z, order='c')
    R = np.asarray(R, order='c')
    is_valid_linkage(Z, throw=True, name='Z')
    is_valid_im(R, throw=True, name='R')

    n = Z.shape[0] + 1
    if Z.shape[0] != R.shape[0]:
        raise ValueError("The inconsistency matrix and linkage matrix each "
                         "have a different number of rows.")
    MI = np.zeros((n - 1,))
    [Z, R] = _copy_arrays_if_base_present([Z, R])
    _hierarchy.get_max_Rfield_for_each_cluster(Z, R, MI, int(n), 3)
    return MI


def maxRstat(Z, R, i):
    """
    Return the maximum statistic for each non-singleton cluster and its
    descendents.

    Parameters
    ----------
    Z : array_like
        The hierarchical clustering encoded as a matrix. See `linkage` for
        more
        information.
    R : array_like
        The inconsistency matrix.
    i : int
        The column of `R` to use as the statistic.

    Returns
    -------
    MR : ndarray
        Calculates the maximum statistic for the i'th column of the
        inconsistency matrix `R` for each non-singleton cluster
        node. ``MR[j]`` is the maximum over ``R[Q(j)-n, i]`` where
        ``Q(j)`` the set of all node ids corresponding to nodes below
        and including ``j``.

    """
    Z = np.asarray(Z, order='c')
    R = np.asarray(R, order='c')
    is_valid_linkage(Z, throw=True, name='Z')
    is_valid_im(R, throw=True, name='R')
    if type(i) is not int:
        raise TypeError('The third argument must be an integer.')
    if i < 0 or i > 3:
        raise ValueError('i must be an integer between 0 and 3 inclusive.')

    if Z.shape[0] != R.shape[0]:
        raise ValueError("The inconsistency matrix and linkage matrix each "
                         "have a different number of rows.")

    n = Z.shape[0] + 1
    MR = np.zeros((n - 1,))
    [Z, R] = _copy_arrays_if_base_present([Z, R])
    _hierarchy.get_max_Rfield_for_each_cluster(Z, R, MR, int(n), i)
    return MR


def leaders(Z, T):
    """
    Return the root nodes in a hierarchical clustering.

    Returns the root nodes in a hierarchical clustering corresponding
    to a cut defined by a flat cluster assignment vector ``T``. See
    the ``fcluster`` function for more information on the format of ``T``.

    For each flat cluster :math:`j` of the :math:`k` flat clusters
    represented in the n-sized flat cluster assignment vector ``T``,
    this function finds the lowest cluster node :math:`i` in the linkage
    tree Z such that:

      * leaf descendents belong only to flat cluster j
        (i.e. ``T[p]==j`` for all :math:`p` in :math:`S(i)` where
        :math:`S(i)` is the set of leaf ids of leaf nodes descendent
        with cluster node :math:`i`)

      * there does not exist a leaf that is not descendent with
        :math:`i` that also belongs to cluster :math:`j`
        (i.e. ``T[q]!=j`` for all :math:`q` not in :math:`S(i)`).  If
        this condition is violated, ``T`` is not a valid cluster
        assignment vector, and an exception will be thrown.

    Parameters
    ----------
    Z : ndarray
        The hierarchical clustering encoded as a matrix. See
        `linkage` for more information.
    T : ndarray
        The flat cluster assignment vector.

    Returns
    -------
    L : ndarray
        The leader linkage node id's stored as a k-element 1-D array
        where ``k`` is the number of flat clusters found in ``T``.

        ``L[j]=i`` is the linkage cluster node id that is the
        leader of flat cluster with id M[j].  If ``i < n``, ``i``
        corresponds to an original observation, otherwise it
        corresponds to a non-singleton cluster.

        For example: if ``L[3]=2`` and ``M[3]=8``, the flat cluster with
        id 8's leader is linkage node 2.
    M : ndarray
        The leader linkage node id's stored as a k-element 1-D array where
        ``k`` is the number of flat clusters found in ``T``. This allows the
        set of flat cluster ids to be any arbitrary set of ``k`` integers.

    """
    Z = np.asarray(Z, order='c')
    T = np.asarray(T, order='c')
    if type(T) != np.ndarray or T.dtype != 'i':
        raise TypeError('T must be a one-dimensional numpy array of
        integers.')
    is_valid_linkage(Z, throw=True, name='Z')
    if len(T) != Z.shape[0] + 1:
        raise ValueError('Mismatch: len(T)!=Z.shape[0] + 1.')

    Cl = np.unique(T)
    kk = len(Cl)
    L = np.zeros((kk,), dtype='i')
    M = np.zeros((kk,), dtype='i')
    n = Z.shape[0] + 1
    [Z, T] = _copy_arrays_if_base_present([Z, T])
    s = _hierarchy.leaders(Z, T, L, M, int(kk), int(n))
    if s >= 0:
        raise ValueError(('T is not a valid assignment vector. Error found '
                          'when examining linkage node %d (< 2n-1).') % s)
    return (L, M)
```