```
Help on module nltk.tree in nltk:

NAME
    nltk.tree

FILE
    /usr/local/lib/python2.7/dist-packages/nltk/tree.py

DESCRIPTION
    Class for representing hierarchical language structures, such as
    syntax trees and morphological trees.

CLASSES
    __builtin__.list(__builtin__.object)
        Tree
            ImmutableTree
                ImmutableMultiParentedTree(ImmutableTree,
MultiParentedTree)
                ImmutableParentedTree(ImmutableTree, ParentedTree)
                ImmutableProbabilisticTree(ImmutableTree,
nltk.probability.ProbabilisticMixIn)
            ProbabilisticTree(Tree, nltk.probability.ProbabilisticMixIn)
    __builtin__.object
        nltk.probability.ProbabilisticMixIn
    AbstractParentedTree(Tree)
        MultiParentedTree
        ParentedTree

    class ImmutableMultiParentedTree(ImmutableTree,
MultiParentedTree)
     |  Method resolution order:
     |      ImmutableMultiParentedTree
     |      ImmutableTree
     |      MultiParentedTree
     |      AbstractParentedTree
     |      Tree
     |      __builtin__.list
     |      __builtin__.object
     |
     |  Methods inherited from ImmutableTree:
     |
     |  __delitem__(self, index)
     |
     |  __delslice__(self, i, j)
     |
     |  __hash__(self)
     |
     |  __iadd__(self, other)
     |
     |  __imul__(self, other)
     |
     |  __init__(self, node_or_str, children=None)
     |
     |  __setitem__(self, index, value)
     |
     |  __setslice__(self, i, j, value)
     |
     |  append(self, v)
     |
     |  extend(self, v)
     |
     |  pop(self, v=None)
     |
     |  remove(self, v)
     |
     |  reverse(self)
     |
     |  sort(self)
     |
     |  ----------------------------------------------------------------------
     |  Data descriptors inherited from ImmutableTree:
     |
     |  node
     |      Get the node value
     |
     |  ----------------------------------------------------------------------
     |  Methods inherited from MultiParentedTree:
     |
     |  left_siblings(self)
     |      A list of all left siblings of this tree, in any of its parent
     |      trees.  A tree may be its own left sibling if it is used as
     |      multiple contiguous children of the same parent.  A tree may
     |      appear multiple times in this list if it is the left sibling
     |      of this tree with respect to multiple parents.
     |
     |      :type: list(MultiParentedTree)
     |
     |  parent_indices(self, parent)
     |      Return a list of the indices where this tree occurs as a child
     |      of ``parent``.  If this child does not occur as a child of
     |      ``parent``, then the empty list is returned.  The following is
     |      always true::
     |
     |        for parent_index in ptree.parent_indices(parent):
     |            parent[parent_index] is ptree
     |
     |  parents(self)
     |      The set of parents of this tree.  If this tree has no parents,
     |      then ``parents`` is the empty set.  To check if a tree is used
     |      as multiple children of the same parent, use the
     |      ``parent_indices()`` method.
     |
     |      :type: list(MultiParentedTree)
     |
     |  right_siblings(self)
     |      A list of all right siblings of this tree, in any of its parent
     |      trees.  A tree may be its own right sibling if it is used as
     |      multiple contiguous children of the same parent.  A tree may
     |      appear multiple times in this list if it is the right sibling
     |      of this tree with respect to multiple parents.
     |
     |      :type: list(MultiParentedTree)
     |
     |  roots(self)
     |      The set of all roots of this tree.  This set is formed by
     |      tracing all possible parent paths until trees with no parents
     |      are found.
     |
     |      :type: list(MultiParentedTree)
     |
```

```
 |  treepositions(self, root)
 |      Return a list of all tree positions that can be used to reach
 |      this multi-parented tree starting from ``root``.  I.e., the
 |      following is always true::
 |
 |        for treepos in ptree.treepositions(root):
 |            root[treepos] is ptree
 |
 |  ------------------------------------------------------------------------
 |  Methods inherited from AbstractParentedTree:
 |
 |  __getslice__(self, start, stop)
 |
 |  insert(self, index, child)
 |
 |  ------------------------------------------------------------------------
 |  Methods inherited from Tree:
 |
 |  __add__(self, v)
 |
 |  __eq__(self, other)
 |
 |  __ge__(self, other)
 |
 |  __getitem__(self, index)
 |
 |  __gt__(self, other)
 |
 |  __le__(self, other)
 |
 |  __lt__(self, other)
 |
 |  __mul__(self, v)
 |
 |  __ne__(self, other)
 |
 |  __radd__(self, v)
 |
 |  __repr__(self)
 |
 |  __rmul__(self, v)
 |
 |  __str__(self)
 |
 |  chomsky_normal_form(self, factor='right', horzMarkov=None,
vertMarkov=0, childChar='|', parentChar='^')
 |      This method can modify a tree in three ways:
 |
 |        1. Convert a tree into its Chomsky Normal Form (CNF)
 |           equivalent -- Every subtree has either two non-terminals
 |           or one terminal as its children.  This process requires
 |           the creation of more"artificial" non-terminal nodes.
 |        2. Markov (vertical) smoothing of children in new artificial
 |           nodes
 |        3. Horizontal (parent) annotation of nodes
 |
 |      :param factor: Right or left factoring method (default = "right")
 |      :type  factor: str = [left|right]
 |      :param horzMarkov: Markov order for sibling smoothing in
artificial nodes (None (default) = include all siblings)
 |      :type  horzMarkov: int | None
 |      :param vertMarkov: Markov order for parent smoothing (0
(default) = no vertical annotation)
 |      :type  vertMarkov: int | None
 |      :param childChar: A string used in construction of the artificial
nodes, separating the head of the
 |                        original subtree from the child nodes that have yet
to be expanded (default = "|")
 |      :type  childChar: str
 |      :param parentChar: A string used to separate the node
representation from its vertical annotation
 |      :type  parentChar: str
 |
 |  collapse_unary(self, collapsePOS=False, collapseRoot=False,
joinChar='+')
 |      Collapse subtrees with a single child (ie. unary productions)
 |      into a new non-terminal (Tree node) joined by 'joinChar'.
 |      This is useful when working with algorithms that do not allow
 |      unary productions, and completely removing the unary
productions
 |      would require loss of useful information.  The Tree is modified
 |      directly (since it is passed by reference) and no value is
returned.
 |
 |      :param collapsePOS: 'False' (default) will not collapse the
parent of leaf nodes (ie.
 |                          Part-of-Speech tags) since they are always unary
productions
 |      :type  collapsePOS: bool
 |      :param collapseRoot: 'False' (default) will not modify the root
production
 |                           if it is unary.  For the Penn WSJ treebank corpus,
this corresponds
 |                           to the TOP -> productions.
 |      :type collapseRoot: bool
 |      :param joinChar: A string used to connect collapsed node
values (default = "+")
 |      :type  joinChar: str
 |
 |  copy(self, deep=False)
 |
 |  draw(self)
 |      Open a new window containing a graphical diagram of this
tree.
 |
 |  flatten(self)
 |      Return a flat version of the tree, with all non-root non-
terminals removed.
 |
 |          >>> t = Tree("(S (NP (D the) (N dog)) (VP (V chased) (NP
(D the) (N cat))))")
 |          >>> print t.flatten()
 |          (S the dog chased the cat)
 |
 |      :return: a tree consisting of this tree's root connected directly
to
 |          its leaves, omitting all intervening non-terminal nodes.
 |      :rtype: Tree
 |
 |  freeze(self, leaf_freezer=None)
```

```
    |
    | height(self)
    |     Return the height of the tree.
    |
    |         >>> t = Tree("(S (NP (D the) (N dog)) (VP (V chased) (NP
(D the) (N cat))))")
    |         >>> t.height()
    |         5
    |         >>> print t[0,0]
    |         (D the)
    |         >>> t[0,0].height()
    |         2
    |
    |     :return: The height of this tree.  The height of a tree
    |         containing no children is 1; the height of a tree
    |         containing only leaves is 2; and the height of any other
    |         tree is one plus the maximum of its children's
    |         heights.
    |     :rtype: int
    |
    | leaf_treeposition(self, index)
    |     :return: The tree position of the ``index``-th leaf in this
    |         tree.  I.e., if ``tp=self.leaf_treeposition(i)``, then
    |         ``self[tp]==self.leaves()[i]``.
    |
    |     :raise IndexError: If this tree contains fewer than ``index+1``
    |         leaves, or if ``index<0``.
    |
    | leaves(self)
    |     Return the leaves of the tree.
    |
    |         >>> t = Tree("(S (NP (D the) (N dog)) (VP (V chased) (NP
(D the) (N cat))))")
    |         >>> t.leaves()
    |         ['the', 'dog', 'chased', 'the', 'cat']
    |
    |     :return: a list containing this tree's leaves.
    |         The order reflects the order of the
    |         leaves in the tree's hierarchical structure.
    |     :rtype: list
    |
    | pos(self)
    |     Return a sequence of pos-tagged words extracted from the
tree.
    |
    |         >>> t = Tree("(S (NP (D the) (N dog)) (VP (V chased) (NP
(D the) (N cat))))")
    |         >>> t.pos()
    |         [('the', 'D'), ('dog', 'N'), ('chased', 'V'), ('the', 'D'), ('cat', 'N')]
    |
    |     :return: a list of tuples containing leaves and pre-terminals
(part-of-speech tags).
    |         The order reflects the order of the leaves in the tree's
hierarchical structure.
    |     :rtype: list(tuple)
    |
    | pprint(self, margin=70, indent=0, nodesep='', parens='()',
quotes=False)
    |     :return: A pretty-printed string representation of this tree.
    |     :rtype: str
    |     :param margin: The right margin at which to do line-wrapping.
    |     :type margin: int
    |     :param indent: The indentation level at which printing
    |         begins.  This number is used to decide how far to indent
    |         subsequent lines.
    |     :type indent: int
    |     :param nodesep: A string that is used to separate the node
    |         from the children.  E.g., the default value ``':'`` gives
    |         trees like ``(S: (NP: I) (VP: (V: saw) (NP: it)))``.
    |
    | pprint_latex_qtree(self)
    |     Returns a representation of the tree compatible with the
    |     LaTeX qtree package. This consists of the string ``\Tree``
    |     followed by the parse tree represented in bracketed notation.
    |
    |     For example, the following result was generated from a parse
tree of
    |     the sentence ``The announcement astounded us``::
    |
    |       \Tree [.I'' [.N'' [.D The ] [.N' [.N announcement ] ] ]
    |           [.I' [.V'' [.V' [.V astounded ] [.N'' [.N' [.N us ] ] ] ] ] ] ]
    |
    |     See http://www.ling.upenn.edu/advice/latex.html for the
LaTeX
    |     style file for the qtree package.
    |
    |     :return: A latex qtree representation of this tree.
    |     :rtype: str
    |
    | productions(self)
    |     Generate the productions that correspond to the non-terminal
nodes of the tree.
    |     For each subtree of the form (P: C1 C2 ... Cn) this produces a
production of the
    |     form P -> C1 C2 ... Cn.
    |
    |         >>> t = Tree("(S (NP (D the) (N dog)) (VP (V chased) (NP
(D the) (N cat))))")
    |         >>> t.productions()
    |         [S -> NP VP, NP -> D N, D -> 'the', N -> 'dog', VP -> V NP,
V -> 'chased',
    |         NP -> D N, D -> 'the', N -> 'cat']
    |
    |     :rtype: list(Production)
    |
    | subtrees(self, filter=None)
    |     Generate all the subtrees of this tree, optionally restricted
    |     to trees matching the filter function.
    |
    |         >>> t = Tree("(S (NP (D the) (N dog)) (VP (V chased) (NP
(D the) (N cat))))")
    |         >>> for s in t.subtrees(lambda t: t.height() == 2):
    |         ...     print s
    |         (D the)
    |         (N dog)
    |         (V chased)
    |         (D the)
    |         (N cat)
    |
    |     :type filter: function
```

```
    |      :param filter: the function to filter all local trees
    |
    |  treeposition_spanning_leaves(self, start, end)
    |      :return: The tree position of the lowest descendant of this
    |         tree that dominates ``self.leaves()[start:end]``.
    |      :raise ValueError: if ``end <= start``
    |
    |  un_chomsky_normal_form(self, expandUnary=True,
childChar='|', parentChar='^', unaryChar='+')
    |      This method modifies the tree in three ways:
    |
    |        1. Transforms a tree in Chomsky Normal Form back to its
    |           original structure (branching greater than two)
    |        2. Removes any parent annotation (if it exists)
    |        3. (optional) expands unary subtrees (if previously
    |           collapsed with collapseUnary(...) )
    |
    |      :param expandUnary: Flag to expand unary or not (default =
True)
    |      :type  expandUnary: bool
    |      :param childChar: A string separating the head node from its
children in an artificial node (default = "|")
    |      :type  childChar: str
    |      :param parentChar: A sting separating the node label from its
parent annotation (default = "^")
    |      :type  parentChar: str
    |      :param unaryChar: A string joining two non-terminals in a
unary production (default = "+")
    |      :type  unaryChar: str
    |
    |  ----------------------------------------------------------------
    |  Class methods inherited from Tree:
    |
    |  convert(cls, tree) from __builtin__.type
    |      Convert a tree between different subtypes of Tree.  ``cls``
determines
    |      which class will be used to encode the new tree.
    |
    |      :type tree: Tree
    |      :param tree: The tree that should be converted.
    |      :return: The new Tree.
    |
    |  parse(cls, s, brackets='()', parse_node=None, parse_leaf=None,
node_pattern=None, leaf_pattern=None,
remove_empty_top_bracketing=False) from __builtin__.type
    |      Parse a bracketed tree string and return the resulting tree.
    |      Trees are represented as nested brackettings, such as::
    |
    |        (S (NP (NNP John)) (VP (V runs)))
    |
    |      :type s: str
    |      :param s: The string to parse
    |
    |      :type brackets: str (length=2)
    |      :param brackets: The bracket characters used to mark the
    |         beginning and end of trees and subtrees.
    |
    |      :type parse_node: function
    |      :type parse_leaf: function
    |      :param parse_node, parse_leaf: If specified, these functions
    |         are applied to the substrings of ``s`` corresponding to
    |         nodes and leaves (respectively) to obtain the values for
    |         those nodes and leaves.  They should have the following
    |         signature:
    |
    |            parse_node(str) -> value
    |
    |         For example, these functions could be used to parse nodes
    |         and leaves whose values should be some type other than
    |         string (such as ``FeatStruct``).
    |         Note that by default, node strings and leaf strings are
    |         delimited by whitespace and brackets; to override this
    |         default, use the ``node_pattern`` and ``leaf_pattern``
    |         arguments.
    |
    |      :type node_pattern: str
    |      :type leaf_pattern: str
    |      :param node_pattern, leaf_pattern: Regular expression
patterns
    |         used to find node and leaf substrings in ``s``.  By
    |         default, both nodes patterns are defined to match any
    |         sequence of non-whitespace non-bracket characters.
    |
    |      :type remove_empty_top_bracketing: bool
    |      :param remove_empty_top_bracketing: If the resulting tree
has
    |         an empty node label, and is length one, then return its
    |         single child instead.  This is useful for treebank trees,
    |         which sometimes contain an extra level of bracketing.
    |
    |      :return: A tree corresponding to the string representation ``s``.
    |         If this class method is called using a subclass of Tree,
    |         then it will return a tree of that type.
    |      :rtype: Tree
    |
    |  ----------------------------------------------------------------
    |  Data descriptors inherited from Tree:
    |
    |  __dict__
    |      dictionary for instance variables (if defined)
    |
    |  __weakref__
    |      list of weak references to the object (if defined)
    |
    |  ----------------------------------------------------------------
    |  Methods inherited from __builtin__.list:
    |
    |  __contains__(...)
    |      x.__contains__(y) <==> y in x
    |
    |  __getattribute__(...)
    |      x.__getattribute__('name') <==> x.name
    |
    |  __iter__(...)
    |      x.__iter__() <==> iter(x)
    |
    |  __len__(...)
    |      x.__len__() <==> len(x)
    |
    |  __reversed__(...)
```

```
    |      L.__reversed__() -- return a reverse iterator over the list
    |
    |  __sizeof__(...)
    |      L.__sizeof__() -- size of L in memory, in bytes
    |
    |  count(...)
    |      L.count(value) -> integer -- return number of occurrences of
value
    |
    |  index(...)
    |      L.index(value, [start, [stop]]) -> integer -- return first index of
value.
    |      Raises ValueError if the value is not present.
    |
    |  ----------------------------------------------------------------------
    |  Data and other attributes inherited from __builtin__.list:
    |
    |  __new__ = <built-in method __new__ of type object>
    |      T.__new__(S, ...) -> a new object with type S, a subtype of T

  class ImmutableParentedTree(ImmutableTree, ParentedTree)
    |  Method resolution order:
    |      ImmutableParentedTree
    |      ImmutableTree
    |      ParentedTree
    |      AbstractParentedTree
    |      Tree
    |      __builtin__.list
    |      __builtin__.object
    |
    |  Methods inherited from ImmutableTree:
    |
    |  __delitem__(self, index)
    |
    |  __delslice__(self, i, j)
    |
    |  __hash__(self)
    |
    |  __iadd__(self, other)
    |
    |  __imul__(self, other)
    |
    |  __init__(self, node_or_str, children=None)
    |
    |  __setitem__(self, index, value)
    |
    |  __setslice__(self, i, j, value)
    |
    |  append(self, v)
    |
    |  extend(self, v)
    |
    |  pop(self, v=None)
    |
    |  remove(self, v)
    |
    |  reverse(self)
    |
    |  sort(self)
    |
    |  ----------------------------------------------------------------------
    |  Data descriptors inherited from ImmutableTree:
    |
    |  node
    |      Get the node value
    |
    |  ----------------------------------------------------------------------
    |  Methods inherited from ParentedTree:
    |
    |  left_sibling(self)
    |      The left sibling of this tree, or None if it has none.
    |
    |  parent(self)
    |      The parent of this tree, or None if it has no parent.
    |
    |  parent_index(self)
    |      The index of this tree in its parent.  I.e.,
    |      ``ptree.parent()[ptree.parent_index()] is ptree``.  Note that
    |      ``ptree.parent_index()`` is not necessarily equal to
    |      ``ptree.parent.index(ptree)``, since the ``index()`` method
    |      returns the first child that is equal to its argument.
    |
    |  right_sibling(self)
    |      The right sibling of this tree, or None if it has none.
    |
    |  root(self)
    |      The root of this tree.  I.e., the unique ancestor of this tree
    |      whose parent is None.  If ``ptree.parent()`` is None, then
    |      ``ptree`` is its own root.
    |
    |  treeposition(self)
    |      The tree position of this tree, relative to the root of the
    |      tree.  I.e., ``ptree.root[ptree.treeposition] is ptree``.
    |
    |  ----------------------------------------------------------------------
    |  Methods inherited from AbstractParentedTree:
    |
    |  __getslice__(self, start, stop)
    |
    |  insert(self, index, child)
    |
    |  ----------------------------------------------------------------------
    |  Methods inherited from Tree:
    |
    |  __add__(self, v)
    |
    |  __eq__(self, other)
    |
    |  __ge__(self, other)
    |
    |  __getitem__(self, index)
    |
    |  __gt__(self, other)
    |
    |  __le__(self, other)
    |
    |  __lt__(self, other)
    |
    |  __mul__(self, v)
    |
```

```
    |  __ne__(self, other)
    |
    |  __radd__(self, v)
    |
    |  __repr__(self)
    |
    |  __rmul__(self, v)
    |
    |  __str__(self)
    |
    |  chomsky_normal_form(self, factor='right', horzMarkov=None,
vertMarkov=0, childChar='|', parentChar='^')
    |      This method can modify a tree in three ways:
    |
    |        1. Convert a tree into its Chomsky Normal Form (CNF)
    |           equivalent -- Every subtree has either two non-terminals
    |           or one terminal as its children.  This process requires
    |           the creation of more"artificial" non-terminal nodes.
    |        2. Markov (vertical) smoothing of children in new artificial
    |           nodes
    |        3. Horizontal (parent) annotation of nodes
    |
    |      :param factor: Right or left factoring method (default = "right")
    |      :type  factor: str = [left|right]
    |      :param horzMarkov: Markov order for sibling smoothing in
artificial nodes (None (default) = include all siblings)
    |      :type  horzMarkov: int | None
    |      :param vertMarkov: Markov order for parent smoothing (0
(default) = no vertical annotation)
    |      :type  vertMarkov: int | None
    |      :param childChar: A string used in construction of the artificial
nodes, separating the head of the
    |                        original subtree from the child nodes that have yet
to be expanded (default = "|")
    |      :type  childChar: str
    |      :param parentChar: A string used to separate the node
representation from its vertical annotation
    |      :type  parentChar: str
    |
    |  collapse_unary(self, collapsePOS=False, collapseRoot=False,
joinChar='+')
    |      Collapse subtrees with a single child (ie. unary productions)
    |      into a new non-terminal (Tree node) joined by 'joinChar'.
    |      This is useful when working with algorithms that do not allow
    |      unary productions, and completely removing the unary
productions
    |      would require loss of useful information.  The Tree is modified
    |      directly (since it is passed by reference) and no value is
returned.
    |
    |      :param collapsePOS: 'False' (default) will not collapse the
parent of leaf nodes (ie.
    |                          Part-of-Speech tags) since they are always unary
productions
    |      :type  collapsePOS: bool
    |      :param collapseRoot: 'False' (default) will not modify the root
production
    |                           if it is unary.  For the Penn WSJ treebank corpus,
this corresponds
    |                           to the TOP -> productions.
    |      :type collapseRoot: bool
    |      :param joinChar: A string used to connect collapsed node
values (default = "+")
    |      :type  joinChar: str
    |
    |  copy(self, deep=False)
    |
    |  draw(self)
    |      Open a new window containing a graphical diagram of this
tree.
    |
    |  flatten(self)
    |      Return a flat version of the tree, with all non-root non-
terminals removed.
    |
    |          >>> t = Tree("(S (NP (D the) (N dog)) (VP (V chased) (NP
(D the) (N cat))))")
    |          >>> print t.flatten()
    |          (S the dog chased the cat)
    |
    |      :return: a tree consisting of this tree's root connected directly
to
    |          its leaves, omitting all intervening non-terminal nodes.
    |      :rtype: Tree
    |
    |  freeze(self, leaf_freezer=None)
    |
    |  height(self)
    |      Return the height of the tree.
    |
    |          >>> t = Tree("(S (NP (D the) (N dog)) (VP (V chased) (NP
(D the) (N cat))))")
    |          >>> t.height()
    |          5
    |          >>> print t[0,0]
    |          (D the)
    |          >>> t[0,0].height()
    |          2
    |
    |      :return: The height of this tree.  The height of a tree
    |          containing no children is 1; the height of a tree
    |          containing only leaves is 2; and the height of any other
    |          tree is one plus the maximum of its children's
    |          heights.
    |      :rtype: int
    |
    |  leaf_treeposition(self, index)
    |      :return: The tree position of the ``index``-th leaf in this
    |          tree.  I.e., if ``tp=self.leaf_treeposition(i)``, then
    |          ``self[tp]==self.leaves()[i]``.
    |
    |      :raise IndexError: If this tree contains fewer than ``index+1``
    |          leaves, or if ``index<0``.
    |
    |  leaves(self)
    |      Return the leaves of the tree.
    |
    |          >>> t = Tree("(S (NP (D the) (N dog)) (VP (V chased) (NP
(D the) (N cat))))")
    |          >>> t.leaves()
```

```
    |         ['the', 'dog', 'chased', 'the', 'cat']
    |
    |     :return: a list containing this tree's leaves.
    |         The order reflects the order of the
    |         leaves in the tree's hierarchical structure.
    |     :rtype: list
    |
    | pos(self)
    |     Return a sequence of pos-tagged words extracted from the
tree.
    |
    |         >>> t = Tree("(S (NP (D the) (N dog)) (VP (V chased) (NP
(D the) (N cat))))")
    |         >>> t.pos()
    |         [('the', 'D'), ('dog', 'N'), ('chased', 'V'), ('the', 'D'), ('cat', 'N')]
    |
    |     :return: a list of tuples containing leaves and pre-terminals
(part-of-speech tags).
    |         The order reflects the order of the leaves in the tree's
hierarchical structure.
    |     :rtype: list(tuple)
    |
    | pprint(self, margin=70, indent=0, nodesep='', parens='()',
quotes=False)
    |     :return: A pretty-printed string representation of this tree.
    |     :rtype: str
    |     :param margin: The right margin at which to do line-wrapping.
    |     :type margin: int
    |     :param indent: The indentation level at which printing
    |         begins.  This number is used to decide how far to indent
    |         subsequent lines.
    |     :type indent: int
    |     :param nodesep: A string that is used to separate the node
    |         from the children.  E.g., the default value ``':'`` gives
    |         trees like ``(S: (NP: I) (VP: (V: saw) (NP: it)))``.
    |
    | pprint_latex_qtree(self)
    |     Returns a representation of the tree compatible with the
    |     LaTeX qtree package. This consists of the string ``\Tree``
    |     followed by the parse tree represented in bracketed notation.
    |
    |     For example, the following result was generated from a parse
tree of
    |     the sentence ``The announcement astounded us``::
    |
    |       \Tree [.I'' [.N'' [.D The ] [.N' [.N announcement ] ] ]
    |           [.I' [.V'' [.V' [.V astounded ] [.N'' [.N' [.N us ] ] ] ] ] ] ]
    |
    |     See http://www.ling.upenn.edu/advice/latex.html for the
LaTeX
    |     style file for the qtree package.
    |
    |     :return: A latex qtree representation of this tree.
    |     :rtype: str
    |
    | productions(self)
    |     Generate the productions that correspond to the non-terminal
nodes of the tree.
    |     For each subtree of the form (P: C1 C2 ... Cn) this produces a
production of the
    |     form P -> C1 C2 ... Cn.
    |
    |         >>> t = Tree("(S (NP (D the) (N dog)) (VP (V chased) (NP
(D the) (N cat))))")
    |         >>> t.productions()
    |         [S -> NP VP, NP -> D N, D -> 'the', N -> 'dog', VP -> V NP,
V -> 'chased',
    |         NP -> D N, D -> 'the', N -> 'cat']
    |
    |     :rtype: list(Production)
    |
    | subtrees(self, filter=None)
    |     Generate all the subtrees of this tree, optionally restricted
    |     to trees matching the filter function.
    |
    |         >>> t = Tree("(S (NP (D the) (N dog)) (VP (V chased) (NP
(D the) (N cat))))")
    |         >>> for s in t.subtrees(lambda t: t.height() == 2):
    |         ...     print s
    |         (D the)
    |         (N dog)
    |         (V chased)
    |         (D the)
    |         (N cat)
    |
    |     :type filter: function
    |     :param filter: the function to filter all local trees
    |
    | treeposition_spanning_leaves(self, start, end)
    |     :return: The tree position of the lowest descendant of this
    |         tree that dominates ``self.leaves()[start:end]``.
    |     :raise ValueError: if ``end <= start``
    |
    | treepositions(self, order='preorder')
    |         >>> t = Tree("(S (NP (D the) (N dog)) (VP (V chased) (NP
(D the) (N cat))))")
    |         >>> t.treepositions() # doctest: +ELLIPSIS
    |         [(), (0,), (0, 0), (0, 0, 0), (0, 1), (0, 1, 0), (1,), (1, 0), (1, 0,
0), ...]
    |         >>> for pos in t.treepositions('leaves'):
    |         ...     t[pos] = t[pos][::-1].upper()
    |         >>> print t
    |         (S (NP (D EHT) (N GOD)) (VP (V DESAHC) (NP (D EHT)
(N TAC))))
    |
    |     :param order: One of: ``preorder``, ``postorder``, ``bothorder``,
    |         ``leaves``.
    |
    | un_chomsky_normal_form(self, expandUnary=True,
childChar='|', parentChar='^', unaryChar='+')
    |     This method modifies the tree in three ways:
    |
    |       1. Transforms a tree in Chomsky Normal Form back to its
    |         original structure (branching greater than two)
    |       2. Removes any parent annotation (if it exists)
    |       3. (optional) expands unary subtrees (if previously
    |         collapsed with collapseUnary(...) )
    |
    |     :param expandUnary: Flag to expand unary or not (default =
True)
```

```
    |      :type  expandUnary: bool
    |      :param childChar: A string separating the head node from its
children in an artificial node (default = "|")
    |      :type  childChar: str
    |      :param parentChar: A sting separating the node label from its
parent annotation (default = "^")
    |      :type  parentChar: str
    |      :param unaryChar: A string joining two non-terminals in a
unary production (default = "+")
    |      :type  unaryChar: str
    |
    | ----------------------------------------------------------------------
    | Class methods inherited from Tree:
    |
    | convert(cls, tree) from __builtin__.type
    |      Convert a tree between different subtypes of Tree.  ``cls``
determines
    |      which class will be used to encode the new tree.
    |
    |      :type tree: Tree
    |      :param tree: The tree that should be converted.
    |      :return: The new Tree.
    |
    | parse(cls, s, brackets='()', parse_node=None, parse_leaf=None,
node_pattern=None, leaf_pattern=None,
remove_empty_top_bracketing=False) from __builtin__.type
    |      Parse a bracketed tree string and return the resulting tree.
    |      Trees are represented as nested brackettings, such as::
    |
    |        (S (NP (NNP John)) (VP (V runs)))
    |
    |      :type s: str
    |      :param s: The string to parse
    |
    |      :type brackets: str (length=2)
    |      :param brackets: The bracket characters used to mark the
    |         beginning and end of trees and subtrees.
    |
    |      :type parse_node: function
    |      :type parse_leaf: function
    |      :param parse_node, parse_leaf: If specified, these functions
    |         are applied to the substrings of ``s`` corresponding to
    |         nodes and leaves (respectively) to obtain the values for
    |         those nodes and leaves.  They should have the following
    |         signature:
    |
    |            parse_node(str) -> value
    |
    |         For example, these functions could be used to parse nodes
    |         and leaves whose values should be some type other than
    |         string (such as ``FeatStruct``).
    |         Note that by default, node strings and leaf strings are
    |         delimited by whitespace and brackets; to override this
    |         default, use the ``node_pattern`` and ``leaf_pattern``
    |         arguments.
    |
    |      :type node_pattern: str
    |      :type leaf_pattern: str
    |      :param node_pattern, leaf_pattern: Regular expression
patterns
    |         used to find node and leaf substrings in ``s``.  By
    |         default, both nodes patterns are defined to match any
    |         sequence of non-whitespace non-bracket characters.
    |
    |      :type remove_empty_top_bracketing: bool
    |      :param remove_empty_top_bracketing: If the resulting tree
has
    |         an empty node label, and is length one, then return its
    |         single child instead.  This is useful for treebank trees,
    |         which sometimes contain an extra level of bracketing.
    |
    |      :return: A tree corresponding to the string representation ``s``.
    |         If this class method is called using a subclass of Tree,
    |         then it will return a tree of that type.
    |      :rtype: Tree
    |
    | ----------------------------------------------------------------------
    | Data descriptors inherited from Tree:
    |
    | __dict__
    |      dictionary for instance variables (if defined)
    |
    | __weakref__
    |      list of weak references to the object (if defined)
    |
    | ----------------------------------------------------------------------
    | Methods inherited from __builtin__.list:
    |
    | __contains__(...)
    |      x.__contains__(y) <==> y in x
    |
    | __getattribute__(...)
    |      x.__getattribute__('name') <==> x.name
    |
    | __iter__(...)
    |      x.__iter__() <==> iter(x)
    |
    | __len__(...)
    |      x.__len__() <==> len(x)
    |
    | __reversed__(...)
    |      L.__reversed__() -- return a reverse iterator over the list
    |
    | __sizeof__(...)
    |      L.__sizeof__() -- size of L in memory, in bytes
    |
    | count(...)
    |      L.count(value) -> integer -- return number of occurrences of
value
    |
    | index(...)
    |      L.index(value, [start, [stop]]) -> integer -- return first index of
value.
    |      Raises ValueError if the value is not present.
    |
    | ----------------------------------------------------------------------
    | Data and other attributes inherited from __builtin__.list:
    |
    | __new__ = <built-in method __new__ of type object>
    |      T.__new__(S, ...) -> a new object with type S, a subtype of T
```

```
class ImmutableProbabilisticTree(ImmutableTree,
nltk.probability.ProbabilisticMixIn)
 | Method resolution order:
 |     ImmutableProbabilisticTree
 |     ImmutableTree
 |     Tree
 |     __builtin__.list
 |     nltk.probability.ProbabilisticMixIn
 |     __builtin__.object
 |
 | Methods defined here:
 |
 | __cmp__(self, other)
 |
 | __eq__(self, other)
 |
 | __init__(self, node_or_str, children=None, **prob_kwargs)
 |
 | __ne__(self, other)
 |
 | __repr__(self)
 |
 | __str__(self)
 |
 | copy(self, deep=False)
 |
 | ----------------------------------------------------------------------
 | Class methods defined here:
 |
 | convert(cls, val) from __builtin__.type
 |
 | ----------------------------------------------------------------------
 | Methods inherited from ImmutableTree:
 |
 | __delitem__(self, index)
 |
 | __delslice__(self, i, j)
 |
 | __hash__(self)
 |
 | __iadd__(self, other)
 |
 | __imul__(self, other)
 |
 | __setitem__(self, index, value)
 |
 | __setslice__(self, i, j, value)
 |
 | append(self, v)
 |
 | extend(self, v)
 |
 | pop(self, v=None)
 |
 | remove(self, v)
 |
 | reverse(self)
 |
 | sort(self)
 |
 | ----------------------------------------------------------------------
 | Data descriptors inherited from ImmutableTree:
 |
 | node
 |     Get the node value
 |
 | ----------------------------------------------------------------------
 | Methods inherited from Tree:
 |
 | __add__(self, v)
 |
 | __ge__(self, other)
 |
 | __getitem__(self, index)
 |
 | __gt__(self, other)
 |
 | __le__(self, other)
 |
 | __lt__(self, other)
 |
 | __mul__(self, v)
 |
 | __radd__(self, v)
 |
 | __rmul__(self, v)
 |
 | chomsky_normal_form(self, factor='right', horzMarkov=None,
vertMarkov=0, childChar='|', parentChar='^')
 |     This method can modify a tree in three ways:
 |
 |       1. Convert a tree into its Chomsky Normal Form (CNF)
 |          equivalent -- Every subtree has either two non-terminals
 |          or one terminal as its children.  This process requires
 |          the creation of more"artificial" non-terminal nodes.
 |       2. Markov (vertical) smoothing of children in new artificial
 |          nodes
 |       3. Horizontal (parent) annotation of nodes
 |
 |     :param factor: Right or left factoring method (default = "right")
 |     :type  factor: str = [left|right]
 |     :param horzMarkov: Markov order for sibling smoothing in
artificial nodes (None (default) = include all siblings)
 |     :type  horzMarkov: int | None
 |     :param vertMarkov: Markov order for parent smoothing (0
(default) = no vertical annotation)
 |     :type  vertMarkov: int | None
 |     :param childChar: A string used in construction of the artificial
nodes, separating the head of the
 |                       original subtree from the child nodes that have yet
to be expanded (default = "|")
 |     :type  childChar: str
 |     :param parentChar: A string used to separate the node
representation from its vertical annotation
 |     :type  parentChar: str
 |
 | collapse_unary(self, collapsePOS=False, collapseRoot=False,
joinChar='+')
 |     Collapse subtrees with a single child (ie. unary productions)
```

```
|      into a new non-terminal (Tree node) joined by 'joinChar'.
|      This is useful when working with algorithms that do not allow
|      unary productions, and completely removing the unary
productions
|      would require loss of useful information.  The Tree is modified
|      directly (since it is passed by reference) and no value is
returned.
|
|      :param collapsePOS: 'False' (default) will not collapse the
parent of leaf nodes (ie.
|                          Part-of-Speech tags) since they are always unary
productions
|      :type  collapsePOS: bool
|      :param collapseRoot: 'False' (default) will not modify the root
production
|                           if it is unary.  For the Penn WSJ treebank corpus,
this corresponds
|                           to the TOP -> productions.
|      :type collapseRoot: bool
|      :param joinChar: A string used to connect collapsed node
values (default = "+")
|      :type  joinChar: str
|
|  draw(self)
|      Open a new window containing a graphical diagram of this
tree.
|
|  flatten(self)
|      Return a flat version of the tree, with all non-root non-
terminals removed.
|
|          >>> t = Tree("(S (NP (D the) (N dog)) (VP (V chased) (NP
(D the) (N cat))))")
|          >>> print t.flatten()
|          (S the dog chased the cat)
|
|      :return: a tree consisting of this tree's root connected directly
to
|          its leaves, omitting all intervening non-terminal nodes.
|      :rtype: Tree
|
|  freeze(self, leaf_freezer=None)
|
|  height(self)
|      Return the height of the tree.
|
|          >>> t = Tree("(S (NP (D the) (N dog)) (VP (V chased) (NP
(D the) (N cat))))")
|          >>> t.height()
|          5
|          >>> print t[0,0]
|          (D the)
|          >>> t[0,0].height()
|          2
|
|      :return: The height of this tree.  The height of a tree
|          containing no children is 1; the height of a tree
|          containing only leaves is 2; and the height of any other
|          tree is one plus the maximum of its children's
|          heights.
|      :rtype: int
|
|  leaf_treeposition(self, index)
|      :return: The tree position of the ``index``-th leaf in this
|          tree.  I.e., if ``tp=self.leaf_treeposition(i)``, then
|          ``self[tp]==self.leaves()[i]``.
|
|      :raise IndexError: If this tree contains fewer than ``index+1``
|          leaves, or if ``index<0``.
|
|  leaves(self)
|      Return the leaves of the tree.
|
|          >>> t = Tree("(S (NP (D the) (N dog)) (VP (V chased) (NP
(D the) (N cat))))")
|          >>> t.leaves()
|          ['the', 'dog', 'chased', 'the', 'cat']
|
|      :return: a list containing this tree's leaves.
|          The order reflects the order of the
|          leaves in the tree's hierarchical structure.
|      :rtype: list
|
|  pos(self)
|      Return a sequence of pos-tagged words extracted from the
tree.
|
|          >>> t = Tree("(S (NP (D the) (N dog)) (VP (V chased) (NP
(D the) (N cat))))")
|          >>> t.pos()
|          [('the', 'D'), ('dog', 'N'), ('chased', 'V'), ('the', 'D'), ('cat', 'N')]
|
|      :return: a list of tuples containing leaves and pre-terminals
(part-of-speech tags).
|          The order reflects the order of the leaves in the tree's
hierarchical structure.
|      :rtype: list(tuple)
|
|  pprint(self, margin=70, indent=0, nodesep='', parens='()',
quotes=False)
|      :return: A pretty-printed string representation of this tree.
|      :rtype: str
|      :param margin: The right margin at which to do line-wrapping.
|      :type margin: int
|      :param indent: The indentation level at which printing
|          begins.  This number is used to decide how far to indent
|          subsequent lines.
|      :type indent: int
|      :param nodesep: A string that is used to separate the node
|          from the children.  E.g., the default value ``':'`` gives
|          trees like ``(S: (NP: I) (VP: (V: saw) (NP: it)))``.
|
|  pprint_latex_qtree(self)
|      Returns a representation of the tree compatible with the
|      LaTeX qtree package. This consists of the string ``\Tree``
|      followed by the parse tree represented in bracketed notation.
|
|      For example, the following result was generated from a parse
tree of
|      the sentence ``The announcement astounded us``::
```

```
    |
    |       \Tree [.I" [.N" [.D The ] [.N' [.N announcement ] ] ]
    |           [.I' [.V" [.V' [.V astounded ] [.N" [.N' [.N us ] ] ] ] ] ] ]
    |
    |     See http://www.ling.upenn.edu/advice/latex.html for the
LaTeX
    |     style file for the qtree package.
    |
    |     :return: A latex qtree representation of this tree.
    |     :rtype: str
    |
    | productions(self)
    |     Generate the productions that correspond to the non-terminal
nodes of the tree.
    |     For each subtree of the form (P: C1 C2 ... Cn) this produces a
production of the
    |     form P -> C1 C2 ... Cn.
    |
    |         >>> t = Tree("(S (NP (D the) (N dog)) (VP (V chased) (NP
(D the) (N cat))))")
    |         >>> t.productions()
    |         [S -> NP VP, NP -> D N, D -> 'the', N -> 'dog', VP -> V NP,
V -> 'chased',
    |         NP -> D N, D -> 'the', N -> 'cat']
    |
    |     :rtype: list(Production)
    |
    | subtrees(self, filter=None)
    |     Generate all the subtrees of this tree, optionally restricted
    |     to trees matching the filter function.
    |
    |         >>> t = Tree("(S (NP (D the) (N dog)) (VP (V chased) (NP
(D the) (N cat))))")
    |         >>> for s in t.subtrees(lambda t: t.height() == 2):
    |         ...     print s
    |         (D the)
    |         (N dog)
    |         (V chased)
    |         (D the)
    |         (N cat)
    |
    |     :type filter: function
    |     :param filter: the function to filter all local trees
    |
    | treeposition_spanning_leaves(self, start, end)
    |     :return: The tree position of the lowest descendant of this
    |         tree that dominates ``self.leaves()[start:end]``.
    |     :raise ValueError: if ``end <= start``
    |
    | treepositions(self, order='preorder')
    |         >>> t = Tree("(S (NP (D the) (N dog)) (VP (V chased) (NP
(D the) (N cat))))")
    |         >>> t.treepositions() # doctest: +ELLIPSIS
    |         [(), (0,), (0, 0), (0, 0, 0), (0, 1), (0, 1, 0), (1,), (1, 0), (1, 0,
0), ...]
    |         >>> for pos in t.treepositions('leaves'):
    |         ...     t[pos] = t[pos][::-1].upper()
    |         >>> print t
    |         (S (NP (D EHT) (N GOD)) (VP (V DESAHC) (NP (D EHT)
(N TAC))))
    |
    |     :param order: One of: ``preorder``, ``postorder``, ``bothorder``,
    |         ``leaves``.
    |
    | un_chomsky_normal_form(self, expandUnary=True,
childChar='|', parentChar='^', unaryChar='+')
    |     This method modifies the tree in three ways:
    |
    |       1. Transforms a tree in Chomsky Normal Form back to its
    |          original structure (branching greater than two)
    |       2. Removes any parent annotation (if it exists)
    |       3. (optional) expands unary subtrees (if previously
    |          collapsed with collapseUnary(...) )
    |
    |     :param expandUnary: Flag to expand unary or not (default =
True)
    |     :type  expandUnary: bool
    |     :param childChar: A string separating the head node from its
children in an artificial node (default = "|")
    |     :type  childChar: str
    |     :param parentChar: A sting separating the node label from its
parent annotation (default = "^")
    |     :type  parentChar: str
    |     :param unaryChar: A string joining two non-terminals in a
unary production (default = "+")
    |     :type  unaryChar: str
    |
    | ----------------------------------------------------------------
    | Class methods inherited from Tree:
    |
    | parse(cls, s, brackets='()', parse_node=None, parse_leaf=None,
node_pattern=None, leaf_pattern=None,
remove_empty_top_bracketing=False) from __builtin__.type
    |     Parse a bracketed tree string and return the resulting tree.
    |     Trees are represented as nested brackettings, such as::
    |
    |       (S (NP (NNP John)) (VP (V runs)))
    |
    |     :type s: str
    |     :param s: The string to parse
    |
    |     :type brackets: str (length=2)
    |     :param brackets: The bracket characters used to mark the
    |         beginning and end of trees and subtrees.
    |
    |     :type parse_node: function
    |     :type parse_leaf: function
    |     :param parse_node, parse_leaf: If specified, these functions
    |         are applied to the substrings of ``s`` corresponding to
    |         nodes and leaves (respectively) to obtain the values for
    |         those nodes and leaves.  They should have the following
    |         signature:
    |
    |            parse_node(str) -> value
    |
    |         For example, these functions could be used to parse nodes
    |         and leaves whose values should be some type other than
    |         string (such as ``FeatStruct``).
    |         Note that by default, node strings and leaf strings are
    |         delimited by whitespace and brackets; to override this
```

```
    |        default, use the ``node_pattern`` and ``leaf_pattern``
    |        arguments.
    |
    |    :type node_pattern: str
    |    :type leaf_pattern: str
    |    :param node_pattern, leaf_pattern: Regular expression
patterns
    |        used to find node and leaf substrings in ``s``.  By
    |        default, both nodes patterns are defined to match any
    |        sequence of non-whitespace non-bracket characters.
    |
    |    :type remove_empty_top_bracketing: bool
    |    :param remove_empty_top_bracketing: If the resulting tree
has
    |        an empty node label, and is length one, then return its
    |        single child instead.  This is useful for treebank trees,
    |        which sometimes contain an extra level of bracketing.
    |
    |    :return: A tree corresponding to the string representation ``s``.
    |        If this class method is called using a subclass of Tree,
    |        then it will return a tree of that type.
    |    :rtype: Tree
    |
    | ----------------------------------------------------------------------
    | Data descriptors inherited from Tree:
    |
    | __dict__
    |    dictionary for instance variables (if defined)
    |
    | __weakref__
    |    list of weak references to the object (if defined)
    |
    | ----------------------------------------------------------------------
    | Methods inherited from __builtin__.list:
    |
    | __contains__(...)
    |    x.__contains__(y) <==> y in x
    |
    | __getattribute__(...)
    |    x.__getattribute__('name') <==> x.name
    |
    | __getslice__(...)
    |    x.__getslice__(i, j) <==> x[i:j]
    |
    |    Use of negative indices is not supported.
    |
    | __iter__(...)
    |    x.__iter__() <==> iter(x)
    |
    | __len__(...)
    |    x.__len__() <==> len(x)
    |
    | __reversed__(...)
    |    L.__reversed__() -- return a reverse iterator over the list
    |
    | __sizeof__(...)
    |    L.__sizeof__() -- size of L in memory, in bytes
    |
    | count(...)
    |    L.count(value) -> integer -- return number of occurrences of
value
    |
    | index(...)
    |    L.index(value, [start, [stop]]) -> integer -- return first index of
value.
    |    Raises ValueError if the value is not present.
    |
    | insert(...)
    |    L.insert(index, object) -- insert object before index
    |
    | ----------------------------------------------------------------------
    | Data and other attributes inherited from __builtin__.list:
    |
    | __new__ = <built-in method __new__ of type object>
    |    T.__new__(S, ...) -> a new object with type S, a subtype of T
    |
    | ----------------------------------------------------------------------
    | Methods inherited from nltk.probability.ProbabilisticMixIn:
    |
    | logprob(self)
    |    Return ``log(p)``, where ``p`` is the probability associated
    |    with this object.
    |
    |    :rtype: float
    |
    | prob(self)
    |    Return the probability associated with this object.
    |
    |    :rtype: float
    |
    | set_logprob(self, logprob)
    |    Set the log probability associated with this object to
    |    ``logprob``.  I.e., set the probability associated with this
    |    object to ``2**(logprob)``.
    |
    |    :param logprob: The new log probability
    |    :type logprob: float
    |
    | set_prob(self, prob)
    |    Set the probability associated with this object to ``prob``.
    |
    |    :param prob: The new probability
    |    :type prob: float

class ImmutableTree(Tree)
    | Method resolution order:
    |    ImmutableTree
    |    Tree
    |    __builtin__.list
    |    __builtin__.object
    |
    | Methods defined here:
    |
    | __delitem__(self, index)
    |
    | __delslice__(self, i, j)
    |
    | __hash__(self)
    |
```

```
 |  __iadd__(self, other)
 |
 |  __imul__(self, other)
 |
 |  __init__(self, node_or_str, children=None)
 |
 |  __setitem__(self, index, value)
 |
 |  __setslice__(self, i, j, value)
 |
 |  append(self, v)
 |
 |  extend(self, v)
 |
 |  pop(self, v=None)
 |
 |  remove(self, v)
 |
 |  reverse(self)
 |
 |  sort(self)
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  node
 |      Get the node value
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from Tree:
 |
 |  __add__(self, v)
 |
 |  __eq__(self, other)
 |
 |  __ge__(self, other)
 |
 |  __getitem__(self, index)
 |
 |  __gt__(self, other)
 |
 |  __le__(self, other)
 |
 |  __lt__(self, other)
 |
 |  __mul__(self, v)
 |
 |  __ne__(self, other)
 |
 |  __radd__(self, v)
 |
 |  __repr__(self)
 |
 |  __rmul__(self, v)
 |
 |  __str__(self)
 |
 |  chomsky_normal_form(self, factor='right', horzMarkov=None,
vertMarkov=0, childChar='|', parentChar='^')
 |      This method can modify a tree in three ways:
 |
 |        1. Convert a tree into its Chomsky Normal Form (CNF)
 |           equivalent -- Every subtree has either two non-terminals
 |           or one terminal as its children.  This process requires
 |           the creation of more"artificial" non-terminal nodes.
 |        2. Markov (vertical) smoothing of children in new artificial
 |           nodes
 |        3. Horizontal (parent) annotation of nodes
 |
 |      :param factor: Right or left factoring method (default = "right")
 |      :type  factor: str = [left|right]
 |      :param horzMarkov: Markov order for sibling smoothing in
artificial nodes (None (default) = include all siblings)
 |      :type  horzMarkov: int | None
 |      :param vertMarkov: Markov order for parent smoothing (0
(default) = no vertical annotation)
 |      :type  vertMarkov: int | None
 |      :param childChar: A string used in construction of the artificial
nodes, separating the head of the
 |                        original subtree from the child nodes that have yet
to be expanded (default = "|")
 |      :type  childChar: str
 |      :param parentChar: A string used to separate the node
representation from its vertical annotation
 |      :type  parentChar: str
 |
 |  collapse_unary(self, collapsePOS=False, collapseRoot=False,
joinChar='+')
 |      Collapse subtrees with a single child (ie. unary productions)
 |      into a new non-terminal (Tree node) joined by 'joinChar'.
 |      This is useful when working with algorithms that do not allow
 |      unary productions, and completely removing the unary
productions
 |      would require loss of useful information.  The Tree is modified
 |      directly (since it is passed by reference) and no value is
returned.
 |
 |      :param collapsePOS: 'False' (default) will not collapse the
parent of leaf nodes (ie.
 |                          Part-of-Speech tags) since they are always unary
productions
 |      :type  collapsePOS: bool
 |      :param collapseRoot: 'False' (default) will not modify the root
production
 |                           if it is unary.  For the Penn WSJ treebank corpus,
this corresponds
 |                           to the TOP -> productions.
 |      :type collapseRoot: bool
 |      :param joinChar: A string used to connect collapsed node
values (default = "+")
 |      :type  joinChar: str
 |
 |  copy(self, deep=False)
 |
 |  draw(self)
 |      Open a new window containing a graphical diagram of this
tree.
 |
 |  flatten(self)
```

```
    |      Return a flat version of the tree, with all non-root non-
terminals removed.
    |
    |          >>> t = Tree("(S (NP (D the) (N dog)) (VP (V chased) (NP
(D the) (N cat))))")
    |          >>> print t.flatten()
    |          (S the dog chased the cat)
    |
    |      :return: a tree consisting of this tree's root connected directly
to
    |          its leaves, omitting all intervening non-terminal nodes.
    |      :rtype: Tree
    |
    |  freeze(self, leaf_freezer=None)
    |
    |  height(self)
    |      Return the height of the tree.
    |
    |          >>> t = Tree("(S (NP (D the) (N dog)) (VP (V chased) (NP
(D the) (N cat))))")
    |          >>> t.height()
    |          5
    |          >>> print t[0,0]
    |          (D the)
    |          >>> t[0,0].height()
    |          2
    |
    |      :return: The height of this tree.  The height of a tree
    |          containing no children is 1; the height of a tree
    |          containing only leaves is 2; and the height of any other
    |          tree is one plus the maximum of its children's
    |          heights.
    |      :rtype: int
    |
    |  leaf_treeposition(self, index)
    |      :return: The tree position of the ``index``-th leaf in this
    |          tree.  I.e., if ``tp=self.leaf_treeposition(i)``, then
    |          ``self[tp]==self.leaves()[i]``.
    |
    |      :raise IndexError: If this tree contains fewer than ``index+1``
    |          leaves, or if ``index<0``.
    |
    |  leaves(self)
    |      Return the leaves of the tree.
    |
    |          >>> t = Tree("(S (NP (D the) (N dog)) (VP (V chased) (NP
(D the) (N cat))))")
    |          >>> t.leaves()
    |          ['the', 'dog', 'chased', 'the', 'cat']
    |
    |      :return: a list containing this tree's leaves.
    |          The order reflects the order of the
    |          leaves in the tree's hierarchical structure.
    |      :rtype: list
    |
    |  pos(self)
    |      Return a sequence of pos-tagged words extracted from the
tree.
    |
    |          >>> t = Tree("(S (NP (D the) (N dog)) (VP (V chased) (NP
(D the) (N cat))))")
    |          >>> t.pos()
    |          [('the', 'D'), ('dog', 'N'), ('chased', 'V'), ('the', 'D'), ('cat', 'N')]
    |
    |      :return: a list of tuples containing leaves and pre-terminals
(part-of-speech tags).
    |          The order reflects the order of the leaves in the tree's
hierarchical structure.
    |      :rtype: list(tuple)
    |
    |  pprint(self, margin=70, indent=0, nodesep='', parens='()',
quotes=False)
    |      :return: A pretty-printed string representation of this tree.
    |      :rtype: str
    |      :param margin: The right margin at which to do line-wrapping.
    |      :type margin: int
    |      :param indent: The indentation level at which printing
    |          begins.  This number is used to decide how far to indent
    |          subsequent lines.
    |      :type indent: int
    |      :param nodesep: A string that is used to separate the node
    |          from the children.  E.g., the default value ``':'`` gives
    |          trees like ``(S: (NP: I) (VP: (V: saw) (NP: it)))``.
    |
    |  pprint_latex_qtree(self)
    |      Returns a representation of the tree compatible with the
    |      LaTeX qtree package. This consists of the string ``\Tree``
    |      followed by the parse tree represented in bracketed notation.
    |
    |      For example, the following result was generated from a parse
tree of
    |      the sentence ``The announcement astounded us``::
    |
    |        \Tree [.I'' [.N'' [.D The ] [.N' [.N announcement ] ] ]
    |            [.I' [.V'' [.V' [.V astounded ] [.N'' [.N' [.N us ] ] ] ] ] ] ]
    |
    |      See http://www.ling.upenn.edu/advice/latex.html for the
LaTeX
    |      style file for the qtree package.
    |
    |      :return: A latex qtree representation of this tree.
    |      :rtype: str
    |
    |  productions(self)
    |      Generate the productions that correspond to the non-terminal
nodes of the tree.
    |      For each subtree of the form (P: C1 C2 ... Cn) this produces a
production of the
    |      form P -> C1 C2 ... Cn.
    |
    |          >>> t = Tree("(S (NP (D the) (N dog)) (VP (V chased) (NP
(D the) (N cat))))")
    |          >>> t.productions()
    |          [S -> NP VP, NP -> D N, D -> 'the', N -> 'dog', VP -> V NP,
V -> 'chased',
    |          NP -> D N, D -> 'the', N -> 'cat']
    |
    |      :rtype: list(Production)
    |
```

```
 |  subtrees(self, filter=None)
 |      Generate all the subtrees of this tree, optionally restricted
 |      to trees matching the filter function.
 |
 |          >>> t = Tree("(S (NP (D the) (N dog)) (VP (V chased) (NP
(D the) (N cat))))")
 |          >>> for s in t.subtrees(lambda t: t.height() == 2):
 |          ...     print s
 |          (D the)
 |          (N dog)
 |          (V chased)
 |          (D the)
 |          (N cat)
 |
 |      :type filter: function
 |      :param filter: the function to filter all local trees
 |
 |  treeposition_spanning_leaves(self, start, end)
 |      :return: The tree position of the lowest descendant of this
 |          tree that dominates ``self.leaves()[start:end]``.
 |      :raise ValueError: if ``end <= start``
 |
 |  treepositions(self, order='preorder')
 |          >>> t = Tree("(S (NP (D the) (N dog)) (VP (V chased) (NP
(D the) (N cat))))")
 |          >>> t.treepositions() # doctest: +ELLIPSIS
 |          [(), (0,), (0, 0), (0, 0, 0), (0, 1), (0, 1, 0), (1,), (1, 0), (1, 0,
0), ...]
 |          >>> for pos in t.treepositions('leaves'):
 |          ...     t[pos] = t[pos][::-1].upper()
 |          >>> print t
 |          (S (NP (D EHT) (N GOD)) (VP (V DESAHC) (NP (D EHT)
(N TAC))))
 |
 |      :param order: One of: ``preorder``, ``postorder``, ``bothorder``,
 |          ``leaves``.
 |
 |  un_chomsky_normal_form(self, expandUnary=True,
childChar='|', parentChar='^', unaryChar='+')
 |      This method modifies the tree in three ways:
 |
 |        1. Transforms a tree in Chomsky Normal Form back to its
 |           original structure (branching greater than two)
 |        2. Removes any parent annotation (if it exists)
 |        3. (optional) expands unary subtrees (if previously
 |           collapsed with collapseUnary(...) )
 |
 |      :param expandUnary: Flag to expand unary or not (default =
True)
 |      :type  expandUnary: bool
 |      :param childChar: A string separating the head node from its
children in an artificial node (default = "|")
 |      :type  childChar: str
 |      :param parentChar: A sting separating the node label from its
parent annotation (default = "^")
 |      :type  parentChar: str
 |      :param unaryChar: A string joining two non-terminals in a
unary production (default = "+")
 |      :type  unaryChar: str
 |
 |  ----------------------------------------------------------------
 |  Class methods inherited from Tree:
 |
 |  convert(cls, tree) from __builtin__.type
 |      Convert a tree between different subtypes of Tree.  ``cls``
determines
 |      which class will be used to encode the new tree.
 |
 |      :type tree: Tree
 |      :param tree: The tree that should be converted.
 |      :return: The new Tree.
 |
 |  parse(cls, s, brackets='()', parse_node=None, parse_leaf=None,
node_pattern=None, leaf_pattern=None,
remove_empty_top_bracketing=False) from __builtin__.type
 |      Parse a bracketed tree string and return the resulting tree.
 |      Trees are represented as nested brackettings, such as::
 |
 |        (S (NP (NNP John)) (VP (V runs)))
 |
 |      :type s: str
 |      :param s: The string to parse
 |
 |      :type brackets: str (length=2)
 |      :param brackets: The bracket characters used to mark the
 |          beginning and end of trees and subtrees.
 |
 |      :type parse_node: function
 |      :type parse_leaf: function
 |      :param parse_node, parse_leaf: If specified, these functions
 |          are applied to the substrings of ``s`` corresponding to
 |          nodes and leaves (respectively) to obtain the values for
 |          those nodes and leaves.  They should have the following
 |          signature:
 |
 |             parse_node(str) -> value
 |
 |          For example, these functions could be used to parse nodes
 |          and leaves whose values should be some type other than
 |          string (such as ``FeatStruct``).
 |          Note that by default, node strings and leaf strings are
 |          delimited by whitespace and brackets; to override this
 |          default, use the ``node_pattern`` and ``leaf_pattern``
 |          arguments.
 |
 |      :type node_pattern: str
 |      :type leaf_pattern: str
 |      :param node_pattern, leaf_pattern: Regular expression
patterns
 |          used to find node and leaf substrings in ``s``.  By
 |          default, both nodes patterns are defined to match any
 |          sequence of non-whitespace non-bracket characters.
 |
 |      :type remove_empty_top_bracketing: bool
 |      :param remove_empty_top_bracketing: If the resulting tree
has
 |          an empty node label, and is length one, then return its
 |          single child instead.  This is useful for treebank trees,
 |          which sometimes contain an extra level of bracketing.
 |
```

```
    |     :return: A tree corresponding to the string representation ``s``.
    |        If this class method is called using a subclass of Tree,
    |        then it will return a tree of that type.
    |     :rtype: Tree
    |
    | ----------------------------------------------------------------------
    | Data descriptors inherited from Tree:
    |
    | __dict__
    |     dictionary for instance variables (if defined)
    |
    | __weakref__
    |     list of weak references to the object (if defined)
    |
    | ----------------------------------------------------------------------
    | Methods inherited from __builtin__.list:
    |
    | __contains__(...)
    |     x.__contains__(y) <==> y in x
    |
    | __getattribute__(...)
    |     x.__getattribute__('name') <==> x.name
    |
    | __getslice__(...)
    |     x.__getslice__(i, j) <==> x[i:j]
    |
    |     Use of negative indices is not supported.
    |
    | __iter__(...)
    |     x.__iter__() <==> iter(x)
    |
    | __len__(...)
    |     x.__len__() <==> len(x)
    |
    | __reversed__(...)
    |     L.__reversed__() -- return a reverse iterator over the list
    |
    | __sizeof__(...)
    |     L.__sizeof__() -- size of L in memory, in bytes
    |
    | count(...)
    |     L.count(value) -> integer -- return number of occurrences of
value
    |
    | index(...)
    |     L.index(value, [start, [stop]]) -> integer -- return first index of
value.
    |     Raises ValueError if the value is not present.
    |
    | insert(...)
    |     L.insert(index, object) -- insert object before index
    |
    | ----------------------------------------------------------------------
    | Data and other attributes inherited from __builtin__.list:
    |
    | __new__ = <built-in method __new__ of type object>
    |     T.__new__(S, ...) -> a new object with type S, a subtype of T

  class MultiParentedTree(AbstractParentedTree)
    | A ``Tree`` that automatically maintains parent pointers for
    | multi-parented trees.  The following are methods for querying
the
    | structure of a multi-parented tree: ``parents()``,
``parent_indices()``,
    | ``left_siblings()``, ``right_siblings()``, ``roots``, ``treepositions``.
    |
    | Each ``MultiParentedTree`` may have zero or more parents.  In
    | particular, subtrees may be shared.  If a single
    | ``MultiParentedTree`` is used as multiple children of the same
    | parent, then that parent will appear multiple times in its
    | ``parents()`` method.
    |
    | ``MultiParentedTrees`` should never be used in the same tree
as
    | ``Trees`` or ``ParentedTrees``.  Mixing tree implementations
may
    | result in incorrect parent pointers and in ``TypeError``
exceptions.
    |
    | Method resolution order:
    |     MultiParentedTree
    |     AbstractParentedTree
    |     Tree
    |     __builtin__.list
    |     __builtin__.object
    |
    | Methods defined here:
    |
    | __init__(self, node_or_str, children=None)
    |
    | left_siblings(self)
    |     A list of all left siblings of this tree, in any of its parent
    |     trees.  A tree may be its own left sibling if it is used as
    |     multiple contiguous children of the same parent.  A tree may
    |     appear multiple times in this list if it is the left sibling
    |     of this tree with respect to multiple parents.
    |
    |     :type: list(MultiParentedTree)
    |
    | parent_indices(self, parent)
    |     Return a list of the indices where this tree occurs as a child
    |     of ``parent``.  If this child does not occur as a child of
    |     ``parent``, then the empty list is returned.  The following is
    |     always true::
    |
    |       for parent_index in ptree.parent_indices(parent):
    |           parent[parent_index] is ptree
    |
    | parents(self)
    |     The set of parents of this tree.  If this tree has no parents,
    |     then ``parents`` is the empty set.  To check if a tree is used
    |     as multiple children of the same parent, use the
    |     ``parent_indices()`` method.
    |
    |     :type: list(MultiParentedTree)
    |
    | right_siblings(self)
    |     A list of all right siblings of this tree, in any of its parent
    |     trees.  A tree may be its own right sibling if it is used as
    |     multiple contiguous children of the same parent.  A tree may
```

```
|    appear multiple times in this list if it is the right sibling
|    of this tree with respect to multiple parents.
|
|    :type: list(MultiParentedTree)
|
| roots(self)
|    The set of all roots of this tree.  This set is formed by
|    tracing all possible parent paths until trees with no parents
|    are found.
|
|    :type: list(MultiParentedTree)
|
| treepositions(self, root)
|    Return a list of all tree positions that can be used to reach
|    this multi-parented tree starting from ``root``.  I.e., the
|    following is always true::
|
|      for treepos in ptree.treepositions(root):
|          root[treepos] is ptree
|
| ------------------------------------------------------------------------
| Methods inherited from AbstractParentedTree:
|
| __delitem__(self, index)
|
| __delslice__(self, start, stop)
|
| __getslice__(self, start, stop)
|
| __setitem__(self, index, value)
|
| __setslice__(self, start, stop, value)
|
| append(self, child)
|
| extend(self, children)
|
| insert(self, index, child)
|
| pop(self, index=-1)
|
| remove(self, child)
|    # n.b.: like `list`, this is done by equality, not identity!
|    # To remove a specific child, use del ptree[i].
|
| ------------------------------------------------------------------------
| Methods inherited from Tree:
|
| __add__(self, v)
|
| __eq__(self, other)
|
| __ge__(self, other)
|
| __getitem__(self, index)
|
| __gt__(self, other)
|
| __le__(self, other)
|
| __lt__(self, other)
|
| __mul__(self, v)
|
| __ne__(self, other)
|
| __radd__(self, v)
|
| __repr__(self)
|
| __rmul__(self, v)
|
| __str__(self)
|
| chomsky_normal_form(self, factor='right', horzMarkov=None,
vertMarkov=0, childChar='|', parentChar='^')
|    This method can modify a tree in three ways:
|
|      1. Convert a tree into its Chomsky Normal Form (CNF)
|         equivalent -- Every subtree has either two non-terminals
|         or one terminal as its children.  This process requires
|         the creation of more"artificial" non-terminal nodes.
|      2. Markov (vertical) smoothing of children in new artificial
|         nodes
|      3. Horizontal (parent) annotation of nodes
|
|    :param factor: Right or left factoring method (default = "right")
|    :type  factor: str = [left|right]
|    :param horzMarkov: Markov order for sibling smoothing in
artificial nodes (None (default) = include all siblings)
|    :type  horzMarkov: int | None
|    :param vertMarkov: Markov order for parent smoothing (0
(default) = no vertical annotation)
|    :type  vertMarkov: int | None
|    :param childChar: A string used in construction of the artificial
nodes, separating the head of the
|                      original subtree from the child nodes that have yet
to be expanded (default = "|")
|    :type  childChar: str
|    :param parentChar: A string used to separate the node
representation from its vertical annotation
|    :type  parentChar: str
|
| collapse_unary(self, collapsePOS=False, collapseRoot=False,
joinChar='+')
|    Collapse subtrees with a single child (ie. unary productions)
|    into a new non-terminal (Tree node) joined by 'joinChar'.
|    This is useful when working with algorithms that do not allow
|    unary productions, and completely removing the unary
productions
|    would require loss of useful information.  The Tree is modified
|    directly (since it is passed by reference) and no value is
returned.
|
|    :param collapsePOS: 'False' (default) will not collapse the
parent of leaf nodes (ie.
|                        Part-of-Speech tags) since they are always unary
productions
|    :type  collapsePOS: bool
```

```
    |      :param collapseRoot: 'False' (default) will not modify the root
production
    |                           if it is unary.  For the Penn WSJ treebank corpus,
this corresponds
    |                           to the TOP -> productions.
    |      :type collapseRoot: bool
    |      :param joinChar: A string used to connect collapsed node
values (default = "+")
    |      :type  joinChar: str
    |
    |  copy(self, deep=False)
    |
    |  draw(self)
    |      Open a new window containing a graphical diagram of this
tree.
    |
    |  flatten(self)
    |      Return a flat version of the tree, with all non-root non-
terminals removed.
    |
    |          >>> t = Tree("(S (NP (D the) (N dog)) (VP (V chased) (NP
(D the) (N cat))))")
    |          >>> print t.flatten()
    |          (S the dog chased the cat)
    |
    |      :return: a tree consisting of this tree's root connected directly
to
    |          its leaves, omitting all intervening non-terminal nodes.
    |      :rtype: Tree
    |
    |  freeze(self, leaf_freezer=None)
    |
    |  height(self)
    |      Return the height of the tree.
    |
    |          >>> t = Tree("(S (NP (D the) (N dog)) (VP (V chased) (NP
(D the) (N cat))))")
    |          >>> t.height()
    |          5
    |          >>> print t[0,0]
    |          (D the)
    |          >>> t[0,0].height()
    |          2
    |
    |      :return: The height of this tree.  The height of a tree
    |          containing no children is 1; the height of a tree
    |          containing only leaves is 2; and the height of any other
    |          tree is one plus the maximum of its children's
    |          heights.
    |      :rtype: int
    |
    |  leaf_treeposition(self, index)
    |      :return: The tree position of the ``index``-th leaf in this
    |          tree.  I.e., if ``tp=self.leaf_treeposition(i)``, then
    |          ``self[tp]==self.leaves()[i]``.
    |
    |      :raise IndexError: If this tree contains fewer than ``index+1``
    |          leaves, or if ``index<0``.
    |
    |  leaves(self)
    |      Return the leaves of the tree.
    |
    |          >>> t = Tree("(S (NP (D the) (N dog)) (VP (V chased) (NP
(D the) (N cat))))")
    |          >>> t.leaves()
    |          ['the', 'dog', 'chased', 'the', 'cat']
    |
    |      :return: a list containing this tree's leaves.
    |          The order reflects the order of the
    |          leaves in the tree's hierarchical structure.
    |      :rtype: list
    |
    |  pos(self)
    |      Return a sequence of pos-tagged words extracted from the
tree.
    |
    |          >>> t = Tree("(S (NP (D the) (N dog)) (VP (V chased) (NP
(D the) (N cat))))")
    |          >>> t.pos()
    |          [('the', 'D'), ('dog', 'N'), ('chased', 'V'), ('the', 'D'), ('cat', 'N')]
    |
    |      :return: a list of tuples containing leaves and pre-terminals
(part-of-speech tags).
    |          The order reflects the order of the leaves in the tree's
hierarchical structure.
    |      :rtype: list(tuple)
    |
    |  pprint(self, margin=70, indent=0, nodesep='', parens='()',
quotes=False)
    |      :return: A pretty-printed string representation of this tree.
    |      :rtype: str
    |      :param margin: The right margin at which to do line-wrapping.
    |      :type margin: int
    |      :param indent: The indentation level at which printing
    |          begins.  This number is used to decide how far to indent
    |          subsequent lines.
    |      :type indent: int
    |      :param nodesep: A string that is used to separate the node
    |          from the children.  E.g., the default value ``':'`` gives
    |          trees like ``(S: (NP: I) (VP: (V: saw) (NP: it)))``.
    |
    |  pprint_latex_qtree(self)
    |      Returns a representation of the tree compatible with the
    |      LaTeX qtree package. This consists of the string ``\Tree``
    |      followed by the parse tree represented in bracketed notation.
    |
    |      For example, the following result was generated from a parse
tree of
    |      the sentence ``The announcement astounded us``::
    |
    |        \Tree [.I'' [.N'' [.D The ] [.N' [.N announcement ] ] ]
    |            [.I' [.V'' [.V' [.V astounded ] [.N'' [.N' [.N us ] ] ] ] ] ] ]
    |
    |      See http://www.ling.upenn.edu/advice/latex.html for the
LaTeX
    |      style file for the qtree package.
    |
    |      :return: A latex qtree representation of this tree.
    |      :rtype: str
    |
```

```
    |  productions(self)
    |      Generate the productions that correspond to the non-terminal
nodes of the tree.
    |      For each subtree of the form (P: C1 C2 ... Cn) this produces a
production of the
    |      form P -> C1 C2 ... Cn.
    |
    |          >>> t = Tree("(S (NP (D the) (N dog)) (VP (V chased) (NP
(D the) (N cat))))")
    |          >>> t.productions()
    |          [S -> NP VP, NP -> D N, D -> 'the', N -> 'dog', VP -> V NP,
V -> 'chased',
    |          NP -> D N, D -> 'the', N -> 'cat']
    |
    |      :rtype: list(Production)
    |
    |  subtrees(self, filter=None)
    |      Generate all the subtrees of this tree, optionally restricted
    |      to trees matching the filter function.
    |
    |          >>> t = Tree("(S (NP (D the) (N dog)) (VP (V chased) (NP
(D the) (N cat))))")
    |          >>> for s in t.subtrees(lambda t: t.height() == 2):
    |          ...     print s
    |          (D the)
    |          (N dog)
    |          (V chased)
    |          (D the)
    |          (N cat)
    |
    |      :type filter: function
    |      :param filter: the function to filter all local trees
    |
    |  treeposition_spanning_leaves(self, start, end)
    |      :return: The tree position of the lowest descendant of this
    |          tree that dominates ``self.leaves()[start:end]``.
    |      :raise ValueError: if ``end <= start``
    |
    |  un_chomsky_normal_form(self, expandUnary=True,
childChar='|', parentChar='^', unaryChar='+')
    |      This method modifies the tree in three ways:
    |
    |        1. Transforms a tree in Chomsky Normal Form back to its
    |           original structure (branching greater than two)
    |        2. Removes any parent annotation (if it exists)
    |        3. (optional) expands unary subtrees (if previously
    |           collapsed with collapseUnary(...) )
    |
    |      :param expandUnary: Flag to expand unary or not (default =
True)
    |      :type  expandUnary: bool
    |      :param childChar: A string separating the head node from its
children in an artificial node (default = "|")
    |      :type  childChar: str
    |      :param parentChar: A sting separating the node label from its
parent annotation (default = "^")
    |      :type  parentChar: str
    |      :param unaryChar: A string joining two non-terminals in a
unary production (default = "+")
    |      :type  unaryChar: str
    |
    |  ----------------------------------------------------------------
    |  Class methods inherited from Tree:
    |
    |  convert(cls, tree) from __builtin__.type
    |      Convert a tree between different subtypes of Tree.  ``cls``
determines
    |      which class will be used to encode the new tree.
    |
    |      :type tree: Tree
    |      :param tree: The tree that should be converted.
    |      :return: The new Tree.
    |
    |  parse(cls, s, brackets='()', parse_node=None, parse_leaf=None,
node_pattern=None, leaf_pattern=None,
remove_empty_top_bracketing=False) from __builtin__.type
    |      Parse a bracketed tree string and return the resulting tree.
    |      Trees are represented as nested brackettings, such as::
    |
    |        (S (NP (NNP John)) (VP (V runs)))
    |
    |      :type s: str
    |      :param s: The string to parse
    |
    |      :type brackets: str (length=2)
    |      :param brackets: The bracket characters used to mark the
    |          beginning and end of trees and subtrees.
    |
    |      :type parse_node: function
    |      :type parse_leaf: function
    |      :param parse_node, parse_leaf: If specified, these functions
    |          are applied to the substrings of ``s`` corresponding to
    |          nodes and leaves (respectively) to obtain the values for
    |          those nodes and leaves.  They should have the following
    |          signature:
    |
    |             parse_node(str) -> value
    |
    |          For example, these functions could be used to parse nodes
    |          and leaves whose values should be some type other than
    |          string (such as ``FeatStruct``).
    |          Note that by default, node strings and leaf strings are
    |          delimited by whitespace and brackets; to override this
    |          default, use the ``node_pattern`` and ``leaf_pattern``
    |          arguments.
    |
    |      :type node_pattern: str
    |      :type leaf_pattern: str
    |      :param node_pattern, leaf_pattern: Regular expression
patterns
    |          used to find node and leaf substrings in ``s``.  By
    |          default, both nodes patterns are defined to match any
    |          sequence of non-whitespace non-bracket characters.
    |
    |      :type remove_empty_top_bracketing: bool
    |      :param remove_empty_top_bracketing: If the resulting tree
has
    |          an empty node label, and is length one, then return its
    |          single child instead.  This is useful for treebank trees,
    |          which sometimes contain an extra level of bracketing.
```

```
    |
    |     :return: A tree corresponding to the string representation ``s``.
    |         If this class method is called using a subclass of Tree,
    |         then it will return a tree of that type.
    |     :rtype: Tree
    |
    | ----------------------------------------------------------------------
    | Data descriptors inherited from Tree:
    |
    | __dict__
    |     dictionary for instance variables (if defined)
    |
    | __weakref__
    |     list of weak references to the object (if defined)
    |
    | ----------------------------------------------------------------------
    | Methods inherited from __builtin__.list:
    |
    | __contains__(...)
    |     x.__contains__(y) <==> y in x
    |
    | __getattribute__(...)
    |     x.__getattribute__('name') <==> x.name
    |
    | __iadd__(...)
    |     x.__iadd__(y) <==> x+=y
    |
    | __imul__(...)
    |     x.__imul__(y) <==> x*=y
    |
    | __iter__(...)
    |     x.__iter__() <==> iter(x)
    |
    | __len__(...)
    |     x.__len__() <==> len(x)
    |
    | __reversed__(...)
    |     L.__reversed__() -- return a reverse iterator over the list
    |
    | __sizeof__(...)
    |     L.__sizeof__() -- size of L in memory, in bytes
    |
    | count(...)
    |     L.count(value) -> integer -- return number of occurrences of
value
    |
    | index(...)
    |     L.index(value, [start, [stop]]) -> integer -- return first index of
value.
    |     Raises ValueError if the value is not present.
    |
    | reverse(...)
    |     L.reverse() -- reverse *IN PLACE*
    |
    | sort(...)
    |     L.sort(cmp=None, key=None, reverse=False) -- stable sort
*IN PLACE*;
    |     cmp(x, y) -> -1, 0, 1
    |
    | ----------------------------------------------------------------------
    | Data and other attributes inherited from __builtin__.list:
    |
    | __hash__ = None
    |
    | __new__ = <built-in method __new__ of type object>
    |     T.__new__(S, ...) -> a new object with type S, a subtype of T

  class ParentedTree(AbstractParentedTree)
    | A ``Tree`` that automatically maintains parent pointers for
    | single-parented trees.  The following are methods for querying
    | the structure of a parented tree: ``parent``, ``parent_index``,
    | ``left_sibling``, ``right_sibling``, ``root``, ``treeposition``.
    |
    | Each ``ParentedTree`` may have at most one parent.  In
    | particular, subtrees may not be shared.  Any attempt to reuse a
    | single ``ParentedTree`` as a child of more than one parent (or
    | as multiple children of the same parent) will cause a
    | ``ValueError`` exception to be raised.
    |
    | ``ParentedTrees`` should never be used in the same tree as
``Trees``
    | or ``MultiParentedTrees``.  Mixing tree implementations may
result
    | in incorrect parent pointers and in ``TypeError`` exceptions.
    |
    | Method resolution order:
    |     ParentedTree
    |     AbstractParentedTree
    |     Tree
    |     __builtin__.list
    |     __builtin__.object
    |
    | Methods defined here:
    |
    | __init__(self, node_or_str, children=None)
    |
    | left_sibling(self)
    |     The left sibling of this tree, or None if it has none.
    |
    | parent(self)
    |     The parent of this tree, or None if it has no parent.
    |
    | parent_index(self)
    |     The index of this tree in its parent.  I.e.,
    |     ``ptree.parent()[ptree.parent_index()] is ptree``.  Note that
    |     ``ptree.parent_index()`` is not necessarily equal to
    |     ``ptree.parent.index(ptree)``, since the ``index()`` method
    |     returns the first child that is equal to its argument.
    |
    | right_sibling(self)
    |     The right sibling of this tree, or None if it has none.
    |
    | root(self)
    |     The root of this tree.  I.e., the unique ancestor of this tree
    |     whose parent is None.  If ``ptree.parent()`` is None, then
    |     ``ptree`` is its own root.
    |
    | treeposition(self)
    |     The tree position of this tree, relative to the root of the
    |     tree.  I.e., ``ptree.root[ptree.treeposition] is ptree``.
```

```
|
| ----------------------------------------------------------------------
| Methods inherited from AbstractParentedTree:
|
| __delitem__(self, index)
|
| __delslice__(self, start, stop)
|
| __getslice__(self, start, stop)
|
| __setitem__(self, index, value)
|
| __setslice__(self, start, stop, value)
|
| append(self, child)
|
| extend(self, children)
|
| insert(self, index, child)
|
| pop(self, index=-1)
|
| remove(self, child)
|     # n.b.: like `list`, this is done by equality, not identity!
|     # To remove a specific child, use del ptree[i].
|
| ----------------------------------------------------------------------
| Methods inherited from Tree:
|
| __add__(self, v)
|
| __eq__(self, other)
|
| __ge__(self, other)
|
| __getitem__(self, index)
|
| __gt__(self, other)
|
| __le__(self, other)
|
| __lt__(self, other)
|
| __mul__(self, v)
|
| __ne__(self, other)
|
| __radd__(self, v)
|
| __repr__(self)
|
| __rmul__(self, v)
|
| __str__(self)
|
| chomsky_normal_form(self, factor='right', horzMarkov=None,
vertMarkov=0, childChar='|', parentChar='^')
|     This method can modify a tree in three ways:
|
|       1. Convert a tree into its Chomsky Normal Form (CNF)
|          equivalent -- Every subtree has either two non-terminals
|          or one terminal as its children.  This process requires
|          the creation of more"artificial" non-terminal nodes.
|       2. Markov (vertical) smoothing of children in new artificial
|          nodes
|       3. Horizontal (parent) annotation of nodes
|
|     :param factor: Right or left factoring method (default = "right")
|     :type  factor: str = [left|right]
|     :param horzMarkov: Markov order for sibling smoothing in
artificial nodes (None (default) = include all siblings)
|     :type  horzMarkov: int | None
|     :param vertMarkov: Markov order for parent smoothing (0
(default) = no vertical annotation)
|     :type  vertMarkov: int | None
|     :param childChar: A string used in construction of the artificial
nodes, separating the head of the
|                       original subtree from the child nodes that have yet
to be expanded (default = "|")
|     :type  childChar: str
|     :param parentChar: A string used to separate the node
representation from its vertical annotation
|     :type  parentChar: str
|
| collapse_unary(self, collapsePOS=False, collapseRoot=False,
joinChar='+')
|     Collapse subtrees with a single child (ie. unary productions)
|     into a new non-terminal (Tree node) joined by 'joinChar'.
|     This is useful when working with algorithms that do not allow
|     unary productions, and completely removing the unary
productions
|     would require loss of useful information.  The Tree is modified
|     directly (since it is passed by reference) and no value is
returned.
|
|     :param collapsePOS: 'False' (default) will not collapse the
parent of leaf nodes (ie.
|                         Part-of-Speech tags) since they are always unary
productions
|     :type  collapsePOS: bool
|     :param collapseRoot: 'False' (default) will not modify the root
production
|                          if it is unary.  For the Penn WSJ treebank corpus,
this corresponds
|                          to the TOP -> productions.
|     :type collapseRoot: bool
|     :param joinChar: A string used to connect collapsed node
values (default = "+")
|     :type  joinChar: str
|
| copy(self, deep=False)
|
| draw(self)
|     Open a new window containing a graphical diagram of this
tree.
|
| flatten(self)
|     Return a flat version of the tree, with all non-root non-
terminals removed.
|
```

```
    |         >>> t = Tree("(S (NP (D the) (N dog)) (VP (V chased) (NP
(D the) (N cat))))")
    |         >>> print t.flatten()
    |         (S the dog chased the cat)
    |
    |     :return: a tree consisting of this tree's root connected directly
to
    |         its leaves, omitting all intervening non-terminal nodes.
    |     :rtype: Tree
    |
    | freeze(self, leaf_freezer=None)
    |
    | height(self)
    |     Return the height of the tree.
    |
    |         >>> t = Tree("(S (NP (D the) (N dog)) (VP (V chased) (NP
(D the) (N cat))))")
    |         >>> t.height()
    |         5
    |         >>> print t[0,0]
    |         (D the)
    |         >>> t[0,0].height()
    |         2
    |
    |     :return: The height of this tree.  The height of a tree
    |         containing no children is 1; the height of a tree
    |         containing only leaves is 2; and the height of any other
    |         tree is one plus the maximum of its children's
    |         heights.
    |     :rtype: int
    |
    | leaf_treeposition(self, index)
    |     :return: The tree position of the ``index``-th leaf in this
    |         tree.  I.e., if ``tp=self.leaf_treeposition(i)``, then
    |         ``self[tp]==self.leaves()[i]``.
    |
    |     :raise IndexError: If this tree contains fewer than ``index+1``
    |         leaves, or if ``index<0``.
    |
    | leaves(self)
    |     Return the leaves of the tree.
    |
    |         >>> t = Tree("(S (NP (D the) (N dog)) (VP (V chased) (NP
(D the) (N cat))))")
    |         >>> t.leaves()
    |         ['the', 'dog', 'chased', 'the', 'cat']
    |
    |     :return: a list containing this tree's leaves.
    |         The order reflects the order of the
    |         leaves in the tree's hierarchical structure.
    |     :rtype: list
    |
    | pos(self)
    |     Return a sequence of pos-tagged words extracted from the
tree.
    |
    |         >>> t = Tree("(S (NP (D the) (N dog)) (VP (V chased) (NP
(D the) (N cat))))")
    |         >>> t.pos()
    |         [('the', 'D'), ('dog', 'N'), ('chased', 'V'), ('the', 'D'), ('cat', 'N')]
    |
    |     :return: a list of tuples containing leaves and pre-terminals
(part-of-speech tags).
    |         The order reflects the order of the leaves in the tree's
hierarchical structure.
    |     :rtype: list(tuple)
    |
    | pprint(self, margin=70, indent=0, nodesep='', parens='()',
quotes=False)
    |     :return: A pretty-printed string representation of this tree.
    |     :rtype: str
    |     :param margin: The right margin at which to do line-wrapping.
    |     :type margin: int
    |     :param indent: The indentation level at which printing
    |         begins.  This number is used to decide how far to indent
    |         subsequent lines.
    |     :type indent: int
    |     :param nodesep: A string that is used to separate the node
    |         from the children.  E.g., the default value ``':'`` gives
    |         trees like ``(S: (NP: I) (VP: (V: saw) (NP: it)))``.
    |
    | pprint_latex_qtree(self)
    |     Returns a representation of the tree compatible with the
    |     LaTeX qtree package. This consists of the string ``\Tree``
    |     followed by the parse tree represented in bracketed notation.
    |
    |     For example, the following result was generated from a parse
tree of
    |     the sentence ``The announcement astounded us``::
    |
    |       \Tree [.I'' [.N'' [.D The ] [.N' [.N announcement ] ] ]
    |           [.I' [.V'' [.V' [.V astounded ] [.N'' [.N' [.N us ] ] ] ] ] ] ]
    |
    |     See http://www.ling.upenn.edu/advice/latex.html for the
LaTeX
    |     style file for the qtree package.
    |
    |     :return: A latex qtree representation of this tree.
    |     :rtype: str
    |
    | productions(self)
    |     Generate the productions that correspond to the non-terminal
nodes of the tree.
    |     For each subtree of the form (P: C1 C2 ... Cn) this produces a
production of the
    |     form P -> C1 C2 ... Cn.
    |
    |         >>> t = Tree("(S (NP (D the) (N dog)) (VP (V chased) (NP
(D the) (N cat))))")
    |         >>> t.productions()
    |         [S -> NP VP, NP -> D N, D -> 'the', N -> 'dog', VP -> V NP,
V -> 'chased',
    |         NP -> D N, D -> 'the', N -> 'cat']
    |
    |     :rtype: list(Production)
    |
    | subtrees(self, filter=None)
    |     Generate all the subtrees of this tree, optionally restricted
    |     to trees matching the filter function.
    |
```

```
    |         >>> t = Tree("(S (NP (D the) (N dog)) (VP (V chased) (NP
(D the) (N cat))))")
    |         >>> for s in t.subtrees(lambda t: t.height() == 2):
    |         ...     print s
    |         (D the)
    |         (N dog)
    |         (V chased)
    |         (D the)
    |         (N cat)
    |
    |     :type filter: function
    |     :param filter: the function to filter all local trees
    |
    | treeposition_spanning_leaves(self, start, end)
    |     :return: The tree position of the lowest descendant of this
    |         tree that dominates ``self.leaves()[start:end]``.
    |     :raise ValueError: if ``end <= start``
    |
    | treepositions(self, order='preorder')
    |         >>> t = Tree("(S (NP (D the) (N dog)) (VP (V chased) (NP
(D the) (N cat))))")
    |         >>> t.treepositions() # doctest: +ELLIPSIS
    |         [(), (0,), (0, 0), (0, 0, 0), (0, 1), (0, 1, 0), (1,), (1, 0), (1, 0,
0), ...]
    |         >>> for pos in t.treepositions('leaves'):
    |         ...     t[pos] = t[pos][::-1].upper()
    |         >>> print t
    |         (S (NP (D EHT) (N GOD)) (VP (V DESAHC) (NP (D EHT)
(N TAC))))
    |
    |     :param order: One of: ``preorder``, ``postorder``, ``bothorder``,
    |         ``leaves``.
    |
    | un_chomsky_normal_form(self, expandUnary=True,
childChar='|', parentChar='^', unaryChar='+')
    |     This method modifies the tree in three ways:
    |
    |       1. Transforms a tree in Chomsky Normal Form back to its
    |          original structure (branching greater than two)
    |       2. Removes any parent annotation (if it exists)
    |       3. (optional) expands unary subtrees (if previously
    |          collapsed with collapseUnary(...) )
    |
    |     :param expandUnary: Flag to expand unary or not (default =
True)
    |     :type  expandUnary: bool
    |     :param childChar: A string separating the head node from its
children in an artificial node (default = "|")
    |     :type  childChar: str
    |     :param parentChar: A sting separating the node label from its
parent annotation (default = "^")
    |     :type  parentChar: str
    |     :param unaryChar: A string joining two non-terminals in a
unary production (default = "+")
    |     :type  unaryChar: str
    |
    | ----------------------------------------------------------------------
    | Class methods inherited from Tree:
    |
    | convert(cls, tree) from __builtin__.type
    |     Convert a tree between different subtypes of Tree.  ``cls``
determines
    |     which class will be used to encode the new tree.
    |
    |     :type tree: Tree
    |     :param tree: The tree that should be converted.
    |     :return: The new Tree.
    |
    | parse(cls, s, brackets='()', parse_node=None, parse_leaf=None,
node_pattern=None, leaf_pattern=None,
remove_empty_top_bracketing=False) from __builtin__.type
    |     Parse a bracketed tree string and return the resulting tree.
    |     Trees are represented as nested brackettings, such as::
    |
    |       (S (NP (NNP John)) (VP (V runs)))
    |
    |     :type s: str
    |     :param s: The string to parse
    |
    |     :type brackets: str (length=2)
    |     :param brackets: The bracket characters used to mark the
    |         beginning and end of trees and subtrees.
    |
    |     :type parse_node: function
    |     :type parse_leaf: function
    |     :param parse_node, parse_leaf: If specified, these functions
    |         are applied to the substrings of ``s`` corresponding to
    |         nodes and leaves (respectively) to obtain the values for
    |         those nodes and leaves.  They should have the following
    |         signature:
    |
    |            parse_node(str) -> value
    |
    |         For example, these functions could be used to parse nodes
    |         and leaves whose values should be some type other than
    |         string (such as ``FeatStruct``).
    |         Note that by default, node strings and leaf strings are
    |         delimited by whitespace and brackets; to override this
    |         default, use the ``node_pattern`` and ``leaf_pattern``
    |         arguments.
    |
    |     :type node_pattern: str
    |     :type leaf_pattern: str
    |     :param node_pattern, leaf_pattern: Regular expression
patterns
    |         used to find node and leaf substrings in ``s``.  By
    |         default, both nodes patterns are defined to match any
    |         sequence of non-whitespace non-bracket characters.
    |
    |     :type remove_empty_top_bracketing: bool
    |     :param remove_empty_top_bracketing: If the resulting tree
has
    |         an empty node label, and is length one, then return its
    |         single child instead.  This is useful for treebank trees,
    |         which sometimes contain an extra level of bracketing.
    |
    |     :return: A tree corresponding to the string representation ``s``.
    |         If this class method is called using a subclass of Tree,
    |         then it will return a tree of that type.
    |     :rtype: Tree
```

```
 |
 | ----------------------------------------------------------------------
 | Data descriptors inherited from Tree:
 |
 | __dict__
 |     dictionary for instance variables (if defined)
 |
 | __weakref__
 |     list of weak references to the object (if defined)
 |
 | ----------------------------------------------------------------------
 | Methods inherited from __builtin__.list:
 |
 | __contains__(...)
 |     x.__contains__(y) <==> y in x
 |
 | __getattribute__(...)
 |     x.__getattribute__('name') <==> x.name
 |
 | __iadd__(...)
 |     x.__iadd__(y) <==> x+=y
 |
 | __imul__(...)
 |     x.__imul__(y) <==> x*=y
 |
 | __iter__(...)
 |     x.__iter__() <==> iter(x)
 |
 | __len__(...)
 |     x.__len__() <==> len(x)
 |
 | __reversed__(...)
 |     L.__reversed__() -- return a reverse iterator over the list
 |
 | __sizeof__(...)
 |     L.__sizeof__() -- size of L in memory, in bytes
 |
 | count(...)
 |     L.count(value) -> integer -- return number of occurrences of
value
 |
 | index(...)
 |     L.index(value, [start, [stop]]) -> integer -- return first index of
value.
 |     Raises ValueError if the value is not present.
 |
 | reverse(...)
 |     L.reverse() -- reverse *IN PLACE*
 |
 | sort(...)
 |     L.sort(cmp=None, key=None, reverse=False) -- stable sort
*IN PLACE*;
 |     cmp(x, y) -> -1, 0, 1
 |
 | ----------------------------------------------------------------------
 | Data and other attributes inherited from __builtin__.list:
 |
 | __hash__ = None
 |
 | __new__ = <built-in method __new__ of type object>
 |     T.__new__(S, ...) -> a new object with type S, a subtype of T

class ProbabilisticMixIn(__builtin__.object)
 | A mix-in class to associate probabilities with other classes
 | (trees, rules, etc.).  To use the ``ProbabilisticMixIn`` class,
 | define a new class that derives from an existing class and from
 | ProbabilisticMixIn.  You will need to define a new constructor for
 | the new class, which explicitly calls the constructors of both its
 | parent classes.  For example:
 |
 |     >>> from nltk.probability import ProbabilisticMixIn
 |     >>> class A:
 |     ...     def __init__(self, x, y): self.data = (x,y)
 |     ...
 |     >>> class ProbabilisticA(A, ProbabilisticMixIn):
 |     ...     def __init__(self, x, y, **prob_kwarg):
 |     ...         A.__init__(self, x, y)
 |     ...         ProbabilisticMixIn.__init__(self, **prob_kwarg)
 |
 | See the documentation for the ProbabilisticMixIn
 | ``constructor<__init__>`` for information about the arguments it
 | expects.
 |
 | You should generally also redefine the string representation
 | methods, the comparison methods, and the hashing method.
 |
 | Methods defined here:
 |
 | __init__(self, **kwargs)
 |     Initialize this object's probability.  This initializer should
 |     be called by subclass constructors.  ``prob`` should generally
be
 |     the first argument for those constructors.
 |
 |     :param prob: The probability associated with the object.
 |     :type prob: float
 |     :param logprob: The log of the probability associated with
 |         the object.
 |     :type logprob: float
 |
 | logprob(self)
 |     Return ``log(p)``, where ``p`` is the probability associated
 |     with this object.
 |
 |     :rtype: float
 |
 | prob(self)
 |     Return the probability associated with this object.
 |
 |     :rtype: float
 |
 | set_logprob(self, logprob)
 |     Set the log probability associated with this object to
 |     ``logprob``.  I.e., set the probability associated with this
 |     object to ``2**(logprob)``.
 |
 |     :param logprob: The new log probability
 |     :type logprob: float
 |
 | set_prob(self, prob)
```

```
|  Set the probability associated with this object to ``prob``.
|
|  :param prob: The new probability
|  :type prob: float
|
| ----------------------------------------------------------------------
| Data descriptors defined here:
|
| __dict__
|  dictionary for instance variables (if defined)
|
| __weakref__
|  list of weak references to the object (if defined)

class ProbabilisticTree(Tree, nltk.probability.ProbabilisticMixIn)
| Method resolution order:
|  ProbabilisticTree
|  Tree
|  __builtin__.list
|  nltk.probability.ProbabilisticMixIn
|  __builtin__.object
|
| Methods defined here:
|
| __cmp__(self, other)
|
| __eq__(self, other)
|
| __init__(self, node_or_str, children=None, **prob_kwargs)
|
| __ne__(self, other)
|
| __repr__(self)
|
| __str__(self)
|
| copy(self, deep=False)
|
| ----------------------------------------------------------------------
| Class methods defined here:
|
| convert(cls, val) from __builtin__.type
|
| ----------------------------------------------------------------------
| Methods inherited from Tree:
|
| __add__(self, v)
|
| __delitem__(self, index)
|
| __ge__(self, other)
|
| __getitem__(self, index)
|
| __gt__(self, other)
|
| __le__(self, other)
|
| __lt__(self, other)
|
| __mul__(self, v)
|
| __radd__(self, v)
|
| __rmul__(self, v)
|
| __setitem__(self, index, value)
|
| chomsky_normal_form(self, factor='right', horzMarkov=None,
vertMarkov=0, childChar='|', parentChar='^')
|  This method can modify a tree in three ways:
|
|  1. Convert a tree into its Chomsky Normal Form (CNF)
|  equivalent -- Every subtree has either two non-terminals
|  or one terminal as its children.  This process requires
|  the creation of more"artificial" non-terminal nodes.
|  2. Markov (vertical) smoothing of children in new artificial
|  nodes
|  3. Horizontal (parent) annotation of nodes
|
|  :param factor: Right or left factoring method (default = "right")
|  :type  factor: str = [left|right]
|  :param horzMarkov: Markov order for sibling smoothing in
artificial nodes (None (default) = include all siblings)
|  :type  horzMarkov: int | None
|  :param vertMarkov: Markov order for parent smoothing (0
(default) = no vertical annotation)
|  :type  vertMarkov: int | None
|  :param childChar: A string used in construction of the artificial
nodes, separating the head of the
|  original subtree from the child nodes that have yet
to be expanded (default = "|")
|  :type  childChar: str
|  :param parentChar: A string used to separate the node
representation from its vertical annotation
|  :type  parentChar: str
|
| collapse_unary(self, collapsePOS=False, collapseRoot=False,
joinChar='+')
|  Collapse subtrees with a single child (ie. unary productions)
|  into a new non-terminal (Tree node) joined by 'joinChar'.
|  This is useful when working with algorithms that do not allow
|  unary productions, and completely removing the unary
productions
|  would require loss of useful information.  The Tree is modified
|  directly (since it is passed by reference) and no value is
returned.
|
|  :param collapsePOS: 'False' (default) will not collapse the
parent of leaf nodes (ie.
|  Part-of-Speech tags) since they are always unary
productions
|  :type  collapsePOS: bool
|  :param collapseRoot: 'False' (default) will not modify the root
production
|  if it is unary.  For the Penn WSJ treebank corpus,
this corresponds
|  to the TOP -> productions.
|  :type collapseRoot: bool
```

```
    |      :param joinChar: A string used to connect collapsed node
values (default = "+")
    |      :type  joinChar: str
    |
    |  draw(self)
    |      Open a new window containing a graphical diagram of this
tree.
    |
    |  flatten(self)
    |      Return a flat version of the tree, with all non-root non-
terminals removed.
    |
    |          >>> t = Tree("(S (NP (D the) (N dog)) (VP (V chased) (NP
(D the) (N cat))))")
    |          >>> print t.flatten()
    |          (S the dog chased the cat)
    |
    |      :return: a tree consisting of this tree's root connected directly
to
    |          its leaves, omitting all intervening non-terminal nodes.
    |      :rtype: Tree
    |
    |  freeze(self, leaf_freezer=None)
    |
    |  height(self)
    |      Return the height of the tree.
    |
    |          >>> t = Tree("(S (NP (D the) (N dog)) (VP (V chased) (NP
(D the) (N cat))))")
    |          >>> t.height()
    |          5
    |          >>> print t[0,0]
    |          (D the)
    |          >>> t[0,0].height()
    |          2
    |
    |      :return: The height of this tree.  The height of a tree
    |          containing no children is 1; the height of a tree
    |          containing only leaves is 2; and the height of any other
    |          tree is one plus the maximum of its children's
    |          heights.
    |      :rtype: int
    |
    |  leaf_treeposition(self, index)
    |      :return: The tree position of the ``index``-th leaf in this
    |          tree.  I.e., if ``tp=self.leaf_treeposition(i)``, then
    |          ``self[tp]==self.leaves()[i]``.
    |
    |      :raise IndexError: If this tree contains fewer than ``index+1``
    |          leaves, or if ``index<0``.
    |
    |  leaves(self)
    |      Return the leaves of the tree.
    |
    |          >>> t = Tree("(S (NP (D the) (N dog)) (VP (V chased) (NP
(D the) (N cat))))")
    |          >>> t.leaves()
    |          ['the', 'dog', 'chased', 'the', 'cat']
    |
    |      :return: a list containing this tree's leaves.
    |          The order reflects the order of the
    |          leaves in the tree's hierarchical structure.
    |      :rtype: list
    |
    |  pos(self)
    |      Return a sequence of pos-tagged words extracted from the
tree.
    |
    |          >>> t = Tree("(S (NP (D the) (N dog)) (VP (V chased) (NP
(D the) (N cat))))")
    |          >>> t.pos()
    |          [('the', 'D'), ('dog', 'N'), ('chased', 'V'), ('the', 'D'), ('cat', 'N')]
    |
    |      :return: a list of tuples containing leaves and pre-terminals
(part-of-speech tags).
    |          The order reflects the order of the leaves in the tree's
hierarchical structure.
    |      :rtype: list(tuple)
    |
    |  pprint(self, margin=70, indent=0, nodesep='', parens='()',
quotes=False)
    |      :return: A pretty-printed string representation of this tree.
    |      :rtype: str
    |      :param margin: The right margin at which to do line-wrapping.
    |      :type margin: int
    |      :param indent: The indentation level at which printing
    |          begins.  This number is used to decide how far to indent
    |          subsequent lines.
    |      :type indent: int
    |      :param nodesep: A string that is used to separate the node
    |          from the children.  E.g., the default value ``':'`` gives
    |          trees like ``(S: (NP: I) (VP: (V: saw) (NP: it)))``.
    |
    |  pprint_latex_qtree(self)
    |      Returns a representation of the tree compatible with the
    |      LaTeX qtree package. This consists of the string ``\Tree``
    |      followed by the parse tree represented in bracketed notation.
    |
    |      For example, the following result was generated from a parse
tree of
    |      the sentence ``The announcement astounded us``::
    |
    |        \Tree [.I'' [.N'' [.D The ] [.N' [.N announcement ] ] ]
    |            [.I' [.V'' [.V' [.V astounded ] [.N'' [.N' [.N us ] ] ] ] ] ] ]
    |
    |      See http://www.ling.upenn.edu/advice/latex.html for the
LaTeX
    |      style file for the qtree package.
    |
    |      :return: A latex qtree representation of this tree.
    |      :rtype: str
    |
    |  productions(self)
    |      Generate the productions that correspond to the non-terminal
nodes of the tree.
    |      For each subtree of the form (P: C1 C2 ... Cn) this produces a
production of the
    |      form P -> C1 C2 ... Cn.
    |
```

```
    |         >>> t = Tree("(S (NP (D the) (N dog)) (VP (V chased) (NP
(D the) (N cat))))")
    |         >>> t.productions()
    |         [S -> NP VP, NP -> D N, D -> 'the', N -> 'dog', VP -> V NP,
V -> 'chased',
    |         NP -> D N, D -> 'the', N -> 'cat']
    |
    |     :rtype: list(Production)
    |
    | subtrees(self, filter=None)
    |     Generate all the subtrees of this tree, optionally restricted
    |     to trees matching the filter function.
    |
    |         >>> t = Tree("(S (NP (D the) (N dog)) (VP (V chased) (NP
(D the) (N cat))))")
    |         >>> for s in t.subtrees(lambda t: t.height() == 2):
    |         ...     print s
    |         (D the)
    |         (N dog)
    |         (V chased)
    |         (D the)
    |         (N cat)
    |
    |     :type filter: function
    |     :param filter: the function to filter all local trees
    |
    | treeposition_spanning_leaves(self, start, end)
    |     :return: The tree position of the lowest descendant of this
    |       tree that dominates ``self.leaves()[start:end]``.
    |     :raise ValueError: if ``end <= start``
    |
    | treepositions(self, order='preorder')
    |         >>> t = Tree("(S (NP (D the) (N dog)) (VP (V chased) (NP
(D the) (N cat))))")
    |         >>> t.treepositions() # doctest: +ELLIPSIS
    |         [(), (0,), (0, 0), (0, 0, 0), (0, 1), (0, 1, 0), (1,), (1, 0), (1, 0,
0), ...]
    |         >>> for pos in t.treepositions('leaves'):
    |         ...     t[pos] = t[pos][::-1].upper()
    |         >>> print t
    |         (S (NP (D EHT) (N GOD)) (VP (V DESAHC) (NP (D EHT)
(N TAC))))
    |
    |     :param order: One of: ``preorder``, ``postorder``, ``bothorder``,
    |       ``leaves``.
    |
    | un_chomsky_normal_form(self, expandUnary=True,
childChar='|', parentChar='^', unaryChar='+')
    |     This method modifies the tree in three ways:
    |
    |       1. Transforms a tree in Chomsky Normal Form back to its
    |         original structure (branching greater than two)
    |       2. Removes any parent annotation (if it exists)
    |       3. (optional) expands unary subtrees (if previously
    |         collapsed with collapseUnary(...) )
    |
    |     :param expandUnary: Flag to expand unary or not (default =
True)
    |     :type  expandUnary: bool
    |     :param childChar: A string separating the head node from its
children in an artificial node (default = "|")
    |     :type  childChar: str
    |     :param parentChar: A sting separating the node label from its
parent annotation (default = "^")
    |     :type  parentChar: str
    |     :param unaryChar: A string joining two non-terminals in a
unary production (default = "+")
    |     :type  unaryChar: str
    |
    | ----------------------------------------------------------------
    | Class methods inherited from Tree:
    |
    | parse(cls, s, brackets='()', parse_node=None, parse_leaf=None,
node_pattern=None, leaf_pattern=None,
remove_empty_top_bracketing=False) from __builtin__.type
    |     Parse a bracketed tree string and return the resulting tree.
    |     Trees are represented as nested brackettings, such as::
    |
    |       (S (NP (NNP John)) (VP (V runs)))
    |
    |     :type s: str
    |     :param s: The string to parse
    |
    |     :type brackets: str (length=2)
    |     :param brackets: The bracket characters used to mark the
    |       beginning and end of trees and subtrees.
    |
    |     :type parse_node: function
    |     :type parse_leaf: function
    |     :param parse_node, parse_leaf: If specified, these functions
    |       are applied to the substrings of ``s`` corresponding to
    |       nodes and leaves (respectively) to obtain the values for
    |       those nodes and leaves.  They should have the following
    |       signature:
    |
    |         parse_node(str) -> value
    |
    |       For example, these functions could be used to parse nodes
    |       and leaves whose values should be some type other than
    |       string (such as ``FeatStruct``).
    |       Note that by default, node strings and leaf strings are
    |       delimited by whitespace and brackets; to override this
    |       default, use the ``node_pattern`` and ``leaf_pattern``
    |       arguments.
    |
    |     :type node_pattern: str
    |     :type leaf_pattern: str
    |     :param node_pattern, leaf_pattern: Regular expression
patterns
    |       used to find node and leaf substrings in ``s``.  By
    |       default, both nodes patterns are defined to match any
    |       sequence of non-whitespace non-bracket characters.
    |
    |     :type remove_empty_top_bracketing: bool
    |     :param remove_empty_top_bracketing: If the resulting tree
has
    |       an empty node label, and is length one, then return its
    |       single child instead.  This is useful for treebank trees,
    |       which sometimes contain an extra level of bracketing.
```

```
|
|    :return: A tree corresponding to the string representation ``s``.
|       If this class method is called using a subclass of Tree,
|       then it will return a tree of that type.
|    :rtype: Tree
|
| ----------------------------------------------------------------------
| Data descriptors inherited from Tree:
|
| __dict__
|    dictionary for instance variables (if defined)
|
| __weakref__
|    list of weak references to the object (if defined)
|
| ----------------------------------------------------------------------
| Methods inherited from __builtin__.list:
|
| __contains__(...)
|    x.__contains__(y) <==> y in x
|
| __delslice__(...)
|    x.__delslice__(i, j) <==> del x[i:j]
|
|    Use of negative indices is not supported.
|
| __getattribute__(...)
|    x.__getattribute__('name') <==> x.name
|
| __getslice__(...)
|    x.__getslice__(i, j) <==> x[i:j]
|
|    Use of negative indices is not supported.
|
| __iadd__(...)
|    x.__iadd__(y) <==> x+=y
|
| __imul__(...)
|    x.__imul__(y) <==> x*=y
|
| __iter__(...)
|    x.__iter__() <==> iter(x)
|
| __len__(...)
|    x.__len__() <==> len(x)
|
| __reversed__(...)
|    L.__reversed__() -- return a reverse iterator over the list
|
| __setslice__(...)
|    x.__setslice__(i, j, y) <==> x[i:j]=y
|
|    Use  of negative indices is not supported.
|
| __sizeof__(...)
|    L.__sizeof__() -- size of L in memory, in bytes
|
| append(...)
|    L.append(object) -- append object to end
|
| count(...)
|    L.count(value) -> integer -- return number of occurrences of
value
|
| extend(...)
|    L.extend(iterable) -- extend list by appending elements from
the iterable
|
| index(...)
|    L.index(value, [start, [stop]]) -> integer -- return first index of
value.
|    Raises ValueError if the value is not present.
|
| insert(...)
|    L.insert(index, object) -- insert object before index
|
| pop(...)
|    L.pop([index]) -> item -- remove and return item at index
(default last).
|    Raises IndexError if list is empty or index is out of range.
|
| remove(...)
|    L.remove(value) -- remove first occurrence of value.
|    Raises ValueError if the value is not present.
|
| reverse(...)
|    L.reverse() -- reverse *IN PLACE*
|
| sort(...)
|    L.sort(cmp=None, key=None, reverse=False) -- stable sort
*IN PLACE*;
|    cmp(x, y) -> -1, 0, 1
|
| ----------------------------------------------------------------------
| Data and other attributes inherited from __builtin__.list:
|
| __hash__ = None
|
| __new__ = <built-in method __new__ of type object>
|    T.__new__(S, ...) -> a new object with type S, a subtype of T
|
| ----------------------------------------------------------------------
| Methods inherited from nltk.probability.ProbabilisticMixIn:
|
| logprob(self)
|    Return ``log(p)``, where ``p`` is the probability associated
|    with this object.
|
|    :rtype: float
|
| prob(self)
|    Return the probability associated with this object.
|
|    :rtype: float
|
| set_logprob(self, logprob)
|    Set the log probability associated with this object to
|    ``logprob``.  I.e., set the probability associated with this
|    object to ``2**(logprob)``.
|
```

```
    |      :param logprob: The new log probability
    |      :type logprob: float
    |
    |  set_prob(self, prob)
    |      Set the probability associated with this object to ``prob``.
    |
    |      :param prob: The new probability
    |      :type prob: float

   class Tree(__builtin__.list)
    |  A Tree represents a hierarchical grouping of leaves and
subtrees.
    |  For example, each constituent in a syntax tree is represented by
a single Tree.
    |
    |  A tree's children are encoded as a list of leaves and subtrees,
    |  where a leaf is a basic (non-tree) value; and a subtree is a
    |  nested Tree.
    |
    |      >>> from nltk.tree import Tree
    |      >>> print Tree(1, [2, Tree(3, [4]), 5])
    |      (1 2 (3 4) 5)
    |      >>> vp = Tree('VP', [Tree('V', ['saw']),
    |      ...                 Tree('NP', ['him'])])
    |      >>> s = Tree('S', [Tree('NP', ['I']), vp])
    |      >>> print s
    |      (S (NP I) (VP (V saw) (NP him)))
    |      >>> print s[1]
    |      (VP (V saw) (NP him))
    |      >>> print s[1,1]
    |      (NP him)
    |      >>> t = Tree("(S (NP I) (VP (V saw) (NP him)))")
    |      >>> s == t
    |      True
    |      >>> t[1][1].node = "X"
    |      >>> print t
    |      (S (NP I) (VP (V saw) (X him)))
    |      >>> t[0], t[1,1] = t[1,1], t[0]
    |      >>> print t
    |      (S (X him) (VP (V saw) (NP I)))
    |
    |  The length of a tree is the number of children it has.
    |
    |      >>> len(t)
    |      2
    |
    |  Any other properties that a Tree defines are known as node
    |  properties, and are used to add information about individual
    |  hierarchical groupings.  For example, syntax trees use a NODE
    |  property to label syntactic constituents with phrase tags, such as
    |  "NP" and "VP".
    |
    |  Several Tree methods use "tree positions" to specify
    |  children or descendants of a tree.  Tree positions are defined as
    |  follows:
    |
    |    - The tree position *i* specifies a Tree's *i*\ th child.
    |    - The tree position ``()`` specifies the Tree itself.
    |    - If *p* is the tree position of descendant *d*, then
    |      *p+i* specifies the *i*\ th child of *d*.
    |
    |  I.e., every tree position is either a single index *i*,
    |  specifying ``tree[i]``; or a sequence *i1, i2, ..., iN*,
    |  specifying ``tree[i1][i2]...[iN]``.
    |
    |  Construct a new tree.  This constructor can be called in one
    |  of two ways:
    |
    |  - ``Tree(node, children)`` constructs a new tree with the
    |      specified node value and list of children.
    |
    |  - ``Tree(s)`` constructs a new tree by parsing the string ``s``.
    |      It is equivalent to calling the class method ``Tree.parse(s)``.
    |
    |  Method resolution order:
    |      Tree
    |      __builtin__.list
    |      __builtin__.object
    |
    |  Methods defined here:
    |
    |  __add__(self, v)
    |
    |  __delitem__(self, index)
    |
    |  __eq__(self, other)
    |
    |  __ge__(self, other)
    |
    |  __getitem__(self, index)
    |
    |  __gt__(self, other)
    |
    |  __init__(self, node_or_str, children=None)
    |
    |  __le__(self, other)
    |
    |  __lt__(self, other)
    |
    |  __mul__(self, v)
    |
    |  __ne__(self, other)
    |
    |  __radd__(self, v)
    |
    |  __repr__(self)
    |
    |  __rmul__(self, v)
    |
    |  __setitem__(self, index, value)
    |
    |  __str__(self)
    |
    |  chomsky_normal_form(self, factor='right', horzMarkov=None,
vertMarkov=0, childChar='|', parentChar='^')
    |      This method can modify a tree in three ways:
    |
    |        1. Convert a tree into its Chomsky Normal Form (CNF)
    |           equivalent -- Every subtree has either two non-terminals
    |           or one terminal as its children.  This process requires
```

```
    |           the creation of more"artificial" non-terminal nodes.
    |        2. Markov (vertical) smoothing of children in new artificial
    |           nodes
    |        3. Horizontal (parent) annotation of nodes
    |
    |     :param factor: Right or left factoring method (default = "right")
    |     :type  factor: str = [left|right]
    |     :param horzMarkov: Markov order for sibling smoothing in
artificial nodes (None (default) = include all siblings)
    |     :type  horzMarkov: int | None
    |     :param vertMarkov: Markov order for parent smoothing (0
(default) = no vertical annotation)
    |     :type  vertMarkov: int | None
    |     :param childChar: A string used in construction of the artificial
nodes, separating the head of the
    |                       original subtree from the child nodes that have yet
to be expanded (default = "|")
    |     :type  childChar: str
    |     :param parentChar: A string used to separate the node
representation from its vertical annotation
    |     :type  parentChar: str
    |
    | collapse_unary(self, collapsePOS=False, collapseRoot=False,
joinChar='+')
    |     Collapse subtrees with a single child (ie. unary productions)
    |     into a new non-terminal (Tree node) joined by 'joinChar'.
    |     This is useful when working with algorithms that do not allow
    |     unary productions, and completely removing the unary
productions
    |     would require loss of useful information.  The Tree is modified
    |     directly (since it is passed by reference) and no value is
returned.
    |
    |     :param collapsePOS: 'False' (default) will not collapse the
parent of leaf nodes (ie.
    |                         Part-of-Speech tags) since they are always unary
productions
    |     :type  collapsePOS: bool
    |     :param collapseRoot: 'False' (default) will not modify the root
production
    |                          if it is unary.  For the Penn WSJ treebank corpus,
this corresponds
    |                          to the TOP -> productions.
    |     :type collapseRoot: bool
    |     :param joinChar: A string used to connect collapsed node
values (default = "+")
    |     :type  joinChar: str
    |
    | copy(self, deep=False)
    |
    | draw(self)
    |     Open a new window containing a graphical diagram of this
tree.
    |
    | flatten(self)
    |     Return a flat version of the tree, with all non-root non-
terminals removed.
    |
    |         >>> t = Tree("(S (NP (D the) (N dog)) (VP (V chased) (NP
(D the) (N cat))))")
    |         >>> print t.flatten()
    |         (S the dog chased the cat)
    |
    |     :return: a tree consisting of this tree's root connected directly
to
    |         its leaves, omitting all intervening non-terminal nodes.
    |     :rtype: Tree
    |
    | freeze(self, leaf_freezer=None)
    |
    | height(self)
    |     Return the height of the tree.
    |
    |         >>> t = Tree("(S (NP (D the) (N dog)) (VP (V chased) (NP
(D the) (N cat))))")
    |         >>> t.height()
    |         5
    |         >>> print t[0,0]
    |         (D the)
    |         >>> t[0,0].height()
    |         2
    |
    |     :return: The height of this tree.  The height of a tree
    |         containing no children is 1; the height of a tree
    |         containing only leaves is 2; and the height of any other
    |         tree is one plus the maximum of its children's
    |         heights.
    |     :rtype: int
    |
    | leaf_treeposition(self, index)
    |     :return: The tree position of the ``index``-th leaf in this
    |         tree.  I.e., if ``tp=self.leaf_treeposition(i)``, then
    |         ``self[tp]==self.leaves()[i]``.
    |
    |     :raise IndexError: If this tree contains fewer than ``index+1``
    |         leaves, or if ``index<0``.
    |
    | leaves(self)
    |     Return the leaves of the tree.
    |
    |         >>> t = Tree("(S (NP (D the) (N dog)) (VP (V chased) (NP
(D the) (N cat))))")
    |         >>> t.leaves()
    |         ['the', 'dog', 'chased', 'the', 'cat']
    |
    |     :return: a list containing this tree's leaves.
    |         The order reflects the order of the
    |         leaves in the tree's hierarchical structure.
    |     :rtype: list
    |
    | pos(self)
    |     Return a sequence of pos-tagged words extracted from the
tree.
    |
    |         >>> t = Tree("(S (NP (D the) (N dog)) (VP (V chased) (NP
(D the) (N cat))))")
    |         >>> t.pos()
    |         [('the', 'D'), ('dog', 'N'), ('chased', 'V'), ('the', 'D'), ('cat', 'N')]
    |
```

```
    |      :return: a list of tuples containing leaves and pre-terminals
(part-of-speech tags).
    |          The order reflects the order of the leaves in the tree's
hierarchical structure.
    |      :rtype: list(tuple)
    |
    |  pprint(self, margin=70, indent=0, nodesep='', parens='()',
quotes=False)
    |      :return: A pretty-printed string representation of this tree.
    |      :rtype: str
    |      :param margin: The right margin at which to do line-wrapping.
    |      :type margin: int
    |      :param indent: The indentation level at which printing
    |          begins.  This number is used to decide how far to indent
    |          subsequent lines.
    |      :type indent: int
    |      :param nodesep: A string that is used to separate the node
    |          from the children.  E.g., the default value ``':'`` gives
    |          trees like ``(S: (NP: I) (VP: (V: saw) (NP: it)))``.
    |
    |  pprint_latex_qtree(self)
    |      Returns a representation of the tree compatible with the
    |      LaTeX qtree package. This consists of the string ``\Tree``
    |      followed by the parse tree represented in bracketed notation.
    |
    |      For example, the following result was generated from a parse
tree of
    |      the sentence ``The announcement astounded us``::
    |
    |        \Tree [.I'' [.N'' [.D The ] [.N' [.N announcement ] ] ]
    |            [.I' [.V'' [.V' [.V astounded ] [.N'' [.N' [.N us ] ] ] ] ] ] ]
    |
    |      See http://www.ling.upenn.edu/advice/latex.html for the
LaTeX
    |      style file for the qtree package.
    |
    |      :return: A latex qtree representation of this tree.
    |      :rtype: str
    |
    |  productions(self)
    |      Generate the productions that correspond to the non-terminal
nodes of the tree.
    |      For each subtree of the form (P: C1 C2 ... Cn) this produces a
production of the
    |      form P -> C1 C2 ... Cn.
    |
    |          >>> t = Tree("(S (NP (D the) (N dog)) (VP (V chased) (NP
(D the) (N cat))))")
    |          >>> t.productions()
    |          [S -> NP VP, NP -> D N, D -> 'the', N -> 'dog', VP -> V NP,
V -> 'chased',
    |          NP -> D N, D -> 'the', N -> 'cat']
    |
    |      :rtype: list(Production)
    |
    |  subtrees(self, filter=None)
    |      Generate all the subtrees of this tree, optionally restricted
    |      to trees matching the filter function.
    |
    |          >>> t = Tree("(S (NP (D the) (N dog)) (VP (V chased) (NP
(D the) (N cat))))")
    |          >>> for s in t.subtrees(lambda t: t.height() == 2):
    |          ...     print s
    |          (D the)
    |          (N dog)
    |          (V chased)
    |          (D the)
    |          (N cat)
    |
    |      :type filter: function
    |      :param filter: the function to filter all local trees
    |
    |  treeposition_spanning_leaves(self, start, end)
    |      :return: The tree position of the lowest descendant of this
    |          tree that dominates ``self.leaves()[start:end]``.
    |      :raise ValueError: if ``end <= start``
    |
    |  treepositions(self, order='preorder')
    |          >>> t = Tree("(S (NP (D the) (N dog)) (VP (V chased) (NP
(D the) (N cat))))")
    |          >>> t.treepositions() # doctest: +ELLIPSIS
    |          [(), (0,), (0, 0), (0, 0, 0), (0, 1), (0, 1, 0), (1,), (1, 0), (1, 0,
0), ...]
    |          >>> for pos in t.treepositions('leaves'):
    |          ...     t[pos] = t[pos][::-1].upper()
    |          >>> print t
    |          (S (NP (D EHT) (N GOD)) (VP (V DESAHC) (NP (D EHT)
(N TAC))))
    |
    |      :param order: One of: ``preorder``, ``postorder``, ``bothorder``,
    |          ``leaves``.
    |
    |  un_chomsky_normal_form(self, expandUnary=True,
childChar='|', parentChar='^', unaryChar='+')
    |      This method modifies the tree in three ways:
    |
    |        1. Transforms a tree in Chomsky Normal Form back to its
    |           original structure (branching greater than two)
    |        2. Removes any parent annotation (if it exists)
    |        3. (optional) expands unary subtrees (if previously
    |           collapsed with collapseUnary(...) )
    |
    |      :param expandUnary: Flag to expand unary or not (default =
True)
    |      :type  expandUnary: bool
    |      :param childChar: A string separating the head node from its
children in an artificial node (default = "|")
    |      :type  childChar: str
    |      :param parentChar: A sting separating the node label from its
parent annotation (default = "^")
    |      :type  parentChar: str
    |      :param unaryChar: A string joining two non-terminals in a
unary production (default = "+")
    |      :type  unaryChar: str
    |
    |  ----------------------------------------------------------------------
    |  Class methods defined here:
    |
    |  convert(cls, tree) from __builtin__.type
```

```
|      Convert a tree between different subtypes of Tree.  ``cls`` determines
|      which class will be used to encode the new tree.
|
|      :type tree: Tree
|      :param tree: The tree that should be converted.
|      :return: The new Tree.
|
|  parse(cls, s, brackets='()', parse_node=None, parse_leaf=None, node_pattern=None, leaf_pattern=None, remove_empty_top_bracketing=False) from __builtin__.type
|      Parse a bracketed tree string and return the resulting tree.
|      Trees are represented as nested brackettings, such as::
|
|        (S (NP (NNP John)) (VP (V runs)))
|
|      :type s: str
|      :param s: The string to parse
|
|      :type brackets: str (length=2)
|      :param brackets: The bracket characters used to mark the
|          beginning and end of trees and subtrees.
|
|      :type parse_node: function
|      :type parse_leaf: function
|      :param parse_node, parse_leaf: If specified, these functions
|          are applied to the substrings of ``s`` corresponding to
|          nodes and leaves (respectively) to obtain the values for
|          those nodes and leaves.  They should have the following
|          signature:
|
|             parse_node(str) -> value
|
|          For example, these functions could be used to parse nodes
|          and leaves whose values should be some type other than
|          string (such as ``FeatStruct``).
|          Note that by default, node strings and leaf strings are
|          delimited by whitespace and brackets; to override this
|          default, use the ``node_pattern`` and ``leaf_pattern``
|          arguments.
|
|      :type node_pattern: str
|      :type leaf_pattern: str
|      :param node_pattern, leaf_pattern: Regular expression patterns
|          used to find node and leaf substrings in ``s``.  By
|          default, both nodes patterns are defined to match any
|          sequence of non-whitespace non-bracket characters.
|
|      :type remove_empty_top_bracketing: bool
|      :param remove_empty_top_bracketing: If the resulting tree has
|          an empty node label, and is length one, then return its
|          single child instead.  This is useful for treebank trees,
|          which sometimes contain an extra level of bracketing.
|
|      :return: A tree corresponding to the string representation ``s``.
|          If this class method is called using a subclass of Tree,
|          then it will return a tree of that type.
|      :rtype: Tree
|
| ----------------------------------------------------------------------
| Data descriptors defined here:
|
| __dict__
|     dictionary for instance variables (if defined)
|
| __weakref__
|     list of weak references to the object (if defined)
|
| ----------------------------------------------------------------------
| Methods inherited from __builtin__.list:
|
| __contains__(...)
|     x.__contains__(y) <==> y in x
|
| __delslice__(...)
|     x.__delslice__(i, j) <==> del x[i:j]
|
|     Use of negative indices is not supported.
|
| __getattribute__(...)
|     x.__getattribute__('name') <==> x.name
|
| __getslice__(...)
|     x.__getslice__(i, j) <==> x[i:j]
|
|     Use of negative indices is not supported.
|
| __iadd__(...)
|     x.__iadd__(y) <==> x+=y
|
| __imul__(...)
|     x.__imul__(y) <==> x*=y
|
| __iter__(...)
|     x.__iter__() <==> iter(x)
|
| __len__(...)
|     x.__len__() <==> len(x)
|
| __reversed__(...)
|     L.__reversed__() -- return a reverse iterator over the list
|
| __setslice__(...)
|     x.__setslice__(i, j, y) <==> x[i:j]=y
|
|     Use  of negative indices is not supported.
|
| __sizeof__(...)
|     L.__sizeof__() -- size of L in memory, in bytes
|
| append(...)
|     L.append(object) -- append object to end
|
| count(...)
|     L.count(value) -> integer -- return number of occurrences of value
|
| extend(...)
```

```
    |      L.extend(iterable) -- extend list by appending elements from
the iterable
    |
    |  index(...)
    |      L.index(value, [start, [stop]]) -> integer -- return first index of
value.
    |      Raises ValueError if the value is not present.
    |
    |  insert(...)
    |      L.insert(index, object) -- insert object before index
    |
    |  pop(...)
    |      L.pop([index]) -> item -- remove and return item at index
(default last).
    |      Raises IndexError if list is empty or index is out of range.
    |
    |  remove(...)
    |      L.remove(value) -- remove first occurrence of value.
    |      Raises ValueError if the value is not present.
    |
    |  reverse(...)
    |      L.reverse() -- reverse *IN PLACE*
    |
    |  sort(...)
    |      L.sort(cmp=None, key=None, reverse=False) -- stable sort
*IN PLACE*;
    |      cmp(x, y) -> -1, 0, 1
    |
    |  ----------------------------------------------------------------------
    |  Data and other attributes inherited from __builtin__.list:
    |
    |  __hash__ = None
    |
    |  __new__ = <built-in method __new__ of type object>
    |      T.__new__(S, ...) -> a new object with type S, a subtype of T

FUNCTIONS
    bracket_parse(s)
        Use Tree.parse(s, remove_empty_top_bracketing=True) instead.

    sinica_parse(s)
        Parse a Sinica Treebank string and return a tree.  Trees are
represented as nested brackettings,
        as shown in the following example (X represents a Chinese
character):

S(goal:NP(Head:Nep:XX)|theme:NP(Head:Nhaa:X)|quantity:Dab:X|H
ead:VL2:X)#0(PERIODCATEGORY)

        :return: A tree corresponding to the string representation.
        :rtype: Tree
        :param s: The string to be converted
        :type s: str

DATA
    __all__ = ['ImmutableProbabilisticTree', 'ImmutableTree',
'Probabilist...
```