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|Head: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...