`tree` Module

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

class `nltk.tree.ImmutableProbabilisticTree`(*node*, *children=None*, ***prob_kwargs*)[source]

Bases: `nltk.tree.ImmutableTree`, `nltk.probability.ProbabilisticMixIn`

classmethod `convert`(*val*)[source]
`copy`(*deep=False*)[source]
`unicode_repr`()

class `nltk.tree.ImmutableTree`(*node*, *children=None*)[source]

Bases: `nltk.tree.Tree`

`append`(*v*)[source]
`extend`(*v*)[source]
`pop`(*v=None*)[source]
`remove`(*v*)[source]
`reverse`()[source]
`set_label`(*value*)[source]

Set the node label. This will only succeed the first time the node label is set, which should occur in ImmutableTree.__init__().

`sort`()[source]

class `nltk.tree.ProbabilisticMixIn`(***kwargs*)

Bases: `builtins.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.

`logprob`()

Return `log(p)`, where `p` is the probability associated with this object.

Return type: float

`prob`()

Return the probability associated with this object.

Return type: float

`set_logprob`(*logprob*)

Set the log probability associated with this object to `logprob`. I.e., set the probability associated with this object to `2**(logprob)`.

Parameters: logprob (*float*) – The new log probability

`set_prob`(*prob*)

Set the probability associated with this object to `prob`.

Parameters: prob (*float*) – The new probability

class `nltk.tree.ProbabilisticTree`(*node*, *children=None*, ***prob_kwargs*)[source]

Bases: `nltk.tree.Tree`, `nltk.probability.ProbabilisticMixIn`

classmethod `convert`(*val*)[source]

`copy`(*deep=False*)[source]

`unicode_repr`()

class `nltk.tree.Tree`(*node*, *children=None*)[source]

Bases: `builtins.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.fromstring("(S (NP I) (VP (V saw) (NP him)))")
```

```
>>> s == t
True
>>> t[1][1].set_label('X')
>>> t[1][1].label()
'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
```

The set_label() and label() methods allow individual constituents to be labeled. For example, syntax trees use this label to specify 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(label, children)` constructs a new tree with the

specified label and list of children.

- `Tree.fromstring(s)` constructs a new tree by parsing the string `s`.

`chomsky_normal_form`(*factor='right', horzMarkov=None, vertMarkov=0, childChar='|', parentChar='^'*)[source]

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

Parameters:

- **factor** (*str = [left|right]*) – Right or left factoring method (default = “right”)
- **horzMarkov** (*int | None*) – Markov order for sibling smoothing in artificial nodes (None (default) = include all siblings)
- **vertMarkov** (*int | None*) – Markov order for parent smoothing (0 (default) = no vertical annotation)

- **childChar** (*str*) – 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 = "|")
- **parentChar** (*str*) – A string used to separate the node representation from its vertical annotation

`collapse_unary`(*collapsePOS=False*, *collapseRoot=False*, *joinChar='+'*)[source]

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.

Parameters:

- **collapsePOS** (*bool*) – 'False' (default) will not collapse the parent of leaf nodes (ie. Part-of-Speech tags) since they are always unary productions
- **collapseRoot** (*bool*) – 'False' (default) will not modify the root production if it is unary. For the Penn WSJ treebank corpus, this corresponds to the TOP -> productions.
- **joinChar** (*str*) – A string used to connect collapsed node values (default = "+")

classmethod `convert`(*tree*)[source]

Convert a tree between different subtypes of Tree. `cls` determines which class will be used to encode the new tree.

Parameters: **tree** (*Tree*) – The tree that should be converted.

Returns: The new Tree.

`copy`(*deep=False*)[source]

`draw`()[source]

Open a new window containing a graphical diagram of this tree.

`flatten`()[source]

Return a flat version of the tree, with all non-root non-terminals removed.

```
>>> t = Tree.fromstring("(S (NP (D the) (N dog)) (VP (V chased) (NP (D the) (N
cat))))")
>>> print(t.flatten())
(S the dog chased the cat)
```

Returns: a tree consisting of this tree's root connected directly to its leaves, omitting all intervening non-terminal nodes.

Return type: Tree

`freeze`(*leaf_freezer=None*)[source]

classmethod `fromstring`(*s*, *brackets='()'*, *read_node=None*, *read_leaf=None*, *node_pattern=None*, *leaf_pattern=None*, *remove_empty_top_bracketing=False*)[source]

Read a bracketed tree string and return the resulting tree. Trees are represented as nested brackettings, such as:

```
(S (NP (NNP John)) (VP (V runs)))
```

- **s** (*str*) – The string to read
- **brackets** (*str (length=2)*) – The bracket characters used to mark the beginning and end of trees and subtrees.
- **read_leaf** (*read_node,*) –

  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:

  read_node(str) -> value

Parameters: For example, these functions could be used to process 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.

- **leaf_pattern** (*node_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.
- **remove_empty_top_bracketing** (*bool*) – 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.

Returns: 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.

Return type: Tree

`height`()[source]

Return the height of the tree.

```
>>> t = Tree.fromstring("(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
```

Returns: 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.

Return type: int

`label`()[source]

Return the node label of the tree.

```
>>> t = Tree.fromstring('(S (NP (D the) (N dog)) (VP (V chased) (NP (D the) (N
cat))))')
>>> t.label()
'S'
```

Returns: the node label (typically a string)

Return type: any

`leaf_treeposition`(*index*)[source]

Returns: 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]`.

Raises IndexError:

If this tree contains fewer than `index+1` leaves, or if `index<0`.

`leaves`()[source]

Return the leaves of the tree.

```
>>> t = Tree.fromstring("(S (NP (D the) (N dog)) (VP (V chased) (NP (D the) (N
cat))))")
>>> t.leaves()
['the', 'dog', 'chased', 'the', 'cat']
```

Returns: a list containing this tree's leaves. The order reflects the order of the leaves in the tree's hierarchical structure.

Return type: list

`node`

Outdated method to access the node value; use the label() method instead.

`pos`()[source]

Return a sequence of pos-tagged words extracted from the tree.

```
>>> t = Tree.fromstring("(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')]
```

Returns: 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.

Return type: list(tuple)

`pprint`(*margin=70, indent=0, nodesep='', parens='()', quotes=False*)[source]

Returns: A pretty-printed string representation of this tree.

Return type: str

Parameters:

- **margin** (*int*) – The right margin at which to do line-wrapping.
- **indent** (*int*) – The indentation level at which printing begins. This number is used to decide how far to indent subsequent lines.
- **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`()[source]

Returns a representation of the tree compatible with the LaTeX qtree package. This consists of the string `\Tree` followed by the 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.

Returns: A latex qtree representation of this tree.

Return type: str

`productions`()[source]

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.fromstring("(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']
```

Return type: list(Production)

`set_label`(*label*)[source]

Set the node label of the tree.

```
>>> t = Tree.fromstring("(S (NP (D the) (N dog)) (VP (V chased) (NP (D the) (N
cat))))")
>>> t.set_label("T")
>>> print(t)
(T (NP (D the) (N dog)) (VP (V chased) (NP (D the) (N cat))))
```

Parameters: label (*any*) – the node label (typically a string)

`subtrees`(*filter=None*)[source]

Generate all the subtrees of this tree, optionally restricted to trees matching the filter function.

```
>>> t = Tree.fromstring("(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)
```

Parameters: filter (*function*) – the function to filter all local trees

`treeposition_spanning_leaves`(*start*, *end*)[source]

Returns: The tree position of the lowest descendant of this tree that dominates `self.leaves()[start:end]`.

Raises ValueError:

if `end <= start`

`treepositions`(*order='preorder'*)[source]

```
>>> t = Tree.fromstring("(S (NP (D the) (N dog)) (VP (V chased) (NP (D the) (N
cat))))")
>>> t.treepositions()
[(), (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))))
```

Parameters: order – One of: `preorder`, `postorder`, `bothorder`, `leaves`.

`un_chomsky_normal_form`(*expandUnary=True*, *childChar='|'*, *parentChar='^'*, *unaryChar='+'*)[source]

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(...))

Parameters:

- **expandUnary** (*bool*) – Flag to expand unary or not (default = True)
- **childChar** (*str*) – A string separating the head node from its children in an artificial node (default = "|")
- **parentChar** (*str*) – A sting separating the node label from its parent annotation (default = "^")
- **unaryChar** (*str*) – A string joining two non-terminals in a unary production (default = "+")

`unicode_repr`()

`nltk.tree.bracket_parse`(*s*)[source]

Use Tree.read(s, remove_empty_top_bracketing=True) instead.

`nltk.tree.sinica_parse`(*s*)[source]

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)

Returns: A tree corresponding to the string representation.

Return type: Tree

Parameters: s (*str*) – The string to be converted

class `nltk.tree.ParentedTree`(*node*, *children=None*)[source]

Bases: `nltk.tree.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.

`left_sibling`()[source]

The left sibling of this tree, or None if it has none.

`parent`()[source]

The parent of this tree, or None if it has no parent.

`parent_index`()[source]

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`()[source]

The right sibling of this tree, or None if it has none.

`root`()[source]

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`()[source]

The tree position of this tree, relative to the root of the tree. I.e., `ptree.root[ptree.treeposition] is ptree`.

class `nltk.tree.MultiParentedTree`(*node*, *children=None*)[source]

Bases: `nltk.tree.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.

left_siblings()[source]

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(*parent*)[source]

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()[source]

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()[source]

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()[source]

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(*root*)[source]

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
```

class nltk.tree.ImmutableParentedTree(*node*, *children=None*)[source]

Bases: nltk.tree.ImmutableTree, nltk.tree.ParentedTree

class nltk.tree.ImmutableMultiParentedTree(*node*, *children=None*)[source]

Bases: nltk.tree.ImmutableTree, nltk.tree.MultiParentedTree