treelib Documentation

Release 1.3.0

Xiaming Chen

Mar 09, 2019

Contents

Redistributed under Apache License (2.0) since version 1.3.0.

Tree data structure is an important data structure in computer programming languages. It has important applications where hierarchical data connections are present such as computer folder structure and decision-tree algorithm in Machine Learning. Thus treelib is created to provide an efficient implementation of tree data structure in Python.

The main features of *treelib* includes:

- Simple to use in both python 2 and 3.
- Efficient operation of node indexing with the benefit of dictionary type.
- Support various tree operations like **traversing**, **insertion**, **deletion**, **node moving**, **shallow/deep copying**, **subtree cutting** etc.
- Support user-defined data payload to accelerate your model construction.
- Has pretty tree showing and text/json dump for pretty show and offline analysis.

Contents:

CHAPTER 1

Install

The rapidest way to install treelib is using the package management tools like `easy_install` or `pip` with command

```
$ sudo easy_install -U treelib
```

or the setup script

```
$ sudo python setup.py install
```

Note: With the package management tools, the hosted version may be falling behind current development branch on Github. If you encounter some problems, try the freshest version on Github or open issues to let me know your problem.

CHAPTER 2

Useful APIs

This *treelib* is a simple module containing only two classes: `Node` and `Tree`. Tree is a self-contained structure with some nodes and connected by branches. One tree has and only has one root, while a node (except root) has several children and merely one parent.

Note: To solve the string compatibility between Python 2.x and 3.x, treelib follows the way of porting Python 3.x to 2/3. That means, all strings are manipulated as unicode and you do not need *u''* prefix anymore. The impacted functions include *str()*, *show()* and *save2file()* routines. But if your data contains non-ascii characters and Python 2.x is used, you have to trigger the compatibility by declaring *unicode_literals* in the code:

```
>>> from __future__ import unicode_literals
```

2.1 Node Objects

class `treelib.`**`Node`**(*[tag[, identifier[, expanded]]]*)

A *Node* object contains basic properties such as node identifier, node tag, parent node, children nodes etc., and some operations for a node.

Class attributes are:

`Node.`**`ADD`**

Addition mode for method *update_fpointer()*.

`Node.`**`DELETE`**

Deletion mode for method *update_fpointer()*.

`Node.`**`INSERT`**

Behave in the same way with Node.ADD since version 1.1.

Instance attributes:

`node.`**`identifier`**

The unique ID of a node within the scope of a tree. This attribute can be accessed and modified with `.` and = operator respectively.

`node.`**`tag`**
The readable node name for human. This attribute can be accessed and modified with `.` and = operator respectively.

`node.`**`bpointer`**
The parent ID of a node. This attribute can be accessed and modified with `.` and = operator respectively.

`node.`**`fpointer`**
With a getting operator, a list of IDs of node's children is obtained. With a setting operator, the value can be list, set, or dict. For list or set, it is converted to a list type by the package; for dict, the keys are treated as the node IDs.

Instance methods:

`node.`**`is_leaf`**`()`
Check if the node has children. Return False if the `fpointer` is empty or None.

`node.`**`is_root`**`()`
Check if the node is the root of present tree.

`node.`**`update_bpointer`**`(`*nid*`)`
Set the parent (indicated by the `nid` parameter) of a node.

`node.`**`update_fpointer`**`(`*nid*, *mode=Node.ADD*`)`
Update the children list with different modes: addition (Node.ADD or Node.INSERT) and deletion (Node.DELETE).

2.2 `Tree` Objects

`class` `node.`**`Tree`**`(`*tree=None*, *deep=False*`)`
The *`Tree`* object defines the tree-like structure based on `Node` objects. A new tree can be created from scratch without any parameter or a shallow/deep copy of another tree. When `deep=True`, a deepcopy operation is performed on feeding `tree` parameter and *more memory is required to create the tree*.

Class attributes are:

`Tree.`**`ROOT`**
Default value for the `level` parameter in tree's methods.

`Tree.`**`DEPTH`**
The depth-first search mode for tree.

`Tree.`**`WIDTH`**
The width-first search mode for tree.

`Tree.`**`ZIGZAG`**
The ZIGZAG search mode for tree.

Instance attributes:

`tree.`**`root`**
Get or set the ID of the root. This attribute can be accessed and modified with `.` and = operator respectively.

Instance methods:

`tree.`**`size`**`()`
Get the number of nodes in this tree.

`tree.`**`contains`**`(`*nid*`)`
Check if the tree contains given node.

`tree.`**`parent`**(*nid*)
: Obtain specific node's parent (Node instance). Return None if the parent is None or does not exist in the tree.

`tree.`**`all_nodes`**()
: Get the list of all the nodes randomly belonging to this tree.

`tree.`**`depth`**()
: Get depth of the tree.

`tree.`**`leaves`**(*nid*)
: Get leaves from given node.

`tree.`**`add_node`**(*node*[, *parent*])
: Add a new node object to the tree and make the parent as the root by default.

`tree.`**`create_node`**(*tag*[, *identifier*[, *parent*]])
: Create a new node and add it to this tree.

`tree.`**`expand_tree`**(*[nid[, mode[, filter[, key[, reverse]]]]]]*)
: Traverse the tree nodes with different modes. `nid` refers to the expanding point to start; `mode` refers to the search mode (Tree.DEPTH, Tree.WIDTH); `filter` refers to the function of one variable to act on the `Node` object; `key`, `reverse` are present to sort :class:Node objects at the same level.

`tree.`**`get_node`**(*nid*)
: Get the object of the node with ID of `nid` An alternative way is using '[]' operation on the tree. But small difference exists between them: the get_node() will return None if `nid` is absent, whereas '[]' will raise `KeyError`.

`tree.`**`is_branch`**(*nid*)
: Get the children (only sons) list of the node with ID == nid.

`tree.`**`siblings`**(*nid*)
: Get all the siblings of given nid.

`tree.`**`move_node`**(*source*, *destination*)
: Move node (source) from its parent to another parent (destination).

`tree.`**`paste`**(*nid*, *new_tree*)
: Paste a new tree to an existing tree, with `nid` becoming the parent of the root of this new tree.

`tree.`**`remove_node`**(*nid*)
: Remove a node and free the memory along with its successors.

`tree.`**`link_past_node`**(*nid*)
: Remove a node and link its children to its parent (root is not allowed).

`tree.`**`rsearch`**(*nid*[, *filter*])
: Search the tree from `nid` to the root along links reservedly. Parameter `filter` refers to the function of one variable to act on the `Node` object.

`tree.`**`show`**(*[nid[, level[, idhidden[, filter[, key[, reverse[, line_type]]]]]]]]*)
: Print the tree structure in hierarchy style. `nid` refers to the expanding point to start; `level` refers to the node level in the tree (root as level 0); `idhidden` refers to hiding the node ID when printing; `filter` refers to the function of one variable to act on the `Node` object; `key`, `reverse` are present to sort `Node` object in the same level.

  You have three ways to output your tree data, i.e., stdout with `show()`, plain text file with `save2file()`, and json string with `to_json()`. The former two use the same backend to generate a string of tree structure in a text graph.

  ***Version >= 1.2.7a*: you can also spicify the `line_type` parameter (now** supporting 'ascii' [default], 'ascii-ex', 'ascii-exr', 'ascii-em', 'ascii-emv', 'ascii-emh') to the change graphical form.

`tree.`**`subtree`**(*nid*)
Return a soft copy of the subtree with `nid` being the root. The softness means all the nodes are shared between subtree and the original.

`tree.`**`remove_subtree`**(*nid*)
Return a subtree with `nid` being the root, and remove all nodes in the subtree from the original one.

`tree.`**`save2file`**(*filename[, nid[, level[, idhidden[, filter[, key[, reverse]]]]]]*)
Save the tree into file for offline analysis.

`tree.`**`to_json`**()
To format the tree in a JSON format.

CHAPTER 3

Examples

3.1 Basic Usage

```
>>> from treelib import Node, Tree
>>> tree = Tree()
>>> tree.create_node("Harry", "harry")  # root node
>>> tree.create_node("Jane", "jane", parent="harry")
>>> tree.create_node("Bill", "bill", parent="harry")
>>> tree.create_node("Diane", "diane", parent="jane")
>>> tree.create_node("Mary", "mary", parent="diane")
>>> tree.create_node("Mark", "mark", parent="jane")
>>> tree.show()
Harry
├── Bill
└── Jane
    ├── Diane
    │   └── Mary
    └── Mark
```

3.2 API Examples

Example 1: Expand a tree with specific mode (Tree.DEPTH [default], Tree.WIDTH, Tree.ZIGZAG).

```
>>> print(','.join([tree[node].tag for node in \
            tree.expand_tree(mode=Tree.DEPTH)]))
Harry,Bill,Jane,Diane,Mary,Mark
```

Example 2: Expand tree with custom filter.

```
>>> print(','.join([tree[node].tag for node in \
            tree.expand_tree(filter = lambda x: \
```

(continues on next page)

(continued from previous page)

```
            x.identifier != 'diane')]))
Harry,Bill,Jane,Mark
```

Example 3: Get a subtree with the root of ‘diane’.

```
>>> sub_t = tree.subtree('diane')
>>> sub_t.show()
Diane
└── Mary
```

Example 4: Paste a new tree to the original one.

```
>>> new_tree = Tree()
>>> new_tree.create_node("n1", 1)  # root node
>>> new_tree.create_node("n2", 2, parent=1)
>>> new_tree.create_node("n3", 3, parent=1)
>>> tree.paste('bill', new_tree)
>>> tree.show()
Harry
├── Bill
│   └── n1
│       ├── n2
│       └── n3
└── Jane
    ├── Diane
    │   └── Mary
    └── Mark
```

Example 5: Remove the existing node from the tree

```
>>> tree.remove_node(1)
>>> tree.show()
Harry
├── Bill
└── Jane
    ├── Diane
    │   └── Mary
    └── Mark
```

Example 6: Move a node to another parent.

```
>>> tree.move_node('mary', 'harry')
>>> tree.show()
Harry
├── Bill
├── Jane
│   ├── Diane
│   └── Mark
└── Mary
```

Example 7: Get the height of the tree.

```
>>> tree.depth()
2
```

Example 8: Get the level of a node.

```
>>> node = tree.get_node("bill")
>>> tree.depth(node)
1
```

Example 9: Print or dump tree structure. For example, the same tree in basic example can be printed with ‘ascii-em’:

```
>>> tree.show(line_type="ascii-em")
Harry
 Bill
 Jane
    Diane
    Mark
 Mary
```

In the JSON form, to_json() takes optional parameter with_data to trigger if the data field is appended into JSON string. For example,

```
>>> print(tree.to_json(with_data=True))
{"Harry": {"data": null, "children": [{"Bill": {"data": null}}, {"Jane": {"data":␣
↪null, "children": [{"Diane": {"data": null}}, {"Mark": {"data": null}}]}}, {"Mary":
↪{"data": null}}]}}
```

3.3 Advanced Usage

Sometimes, you need trees to store your own data. The newsest version of *treelib* supports `.data` variable to store whatever you want. For example, to define a flower tree with your own data:

```
>>> class Flower(object): \
        def __init__(self, color): \
            self.color = color
```

You can create a flower tree now:

```
>>> ftree = Tree()
>>> ftree.create_node("Root", "root")
>>> ftree.create_node("F1", "f1", parent='root', data=Flower("white"))
>>> ftree.create_node("F2", "f2", parent='root', data=Flower("red"))
```

Notes: Before version 1.2.5, you may need to inherit and modify the behaviors of tree. Both are supported since then. For flower example,

```
>>> class FlowerNode(treelib.Node): \
        def __init__(self, color): \
            self.color = color
>>> # create a new node
>>> fnode = FlowerNode("white")
```

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index