treelib Documentation

Release 1.5.5

Xiaming Chen

Mar 09, 2019

Contents:

CHAPTER 1

Introduction

Tree is an important data structure in computer science. Examples are shown in ML algorithm designs such as random forest tree and software engineering such as file system index. treelib is created to provide an efficient implementation of tree data structure in Python.

The main features of *treelib* includes:

- Efficient operation of node searching, O(1).
- Support common tree operations like **traversing**, **insertion**, **deletion**, **node moving**, **shallow/deep copying**, **subtree cutting** etc.
- Support user-defined data payload to accelerate your model construction.
- Pretty tree showing and text/json dump for pretty show and offline analysis.
- Compatible with Python 2 and 3.

CHAPTER 2

Installation

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 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}}]}}
```

Example 10: Save tree into file The function save2file require a filename. The file is opened to write using mode ‘ab’.

```
>>> tree.save2file('tree.txt')
```

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

Printing the colors of the tree:

```
>>> ftree.show(data_property="color")
    black
    ├── white
    └── 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