binary$_{t}reeDocumentation$

Release 0.1.0

Han Keong

Apr 30, 2018

Contents

Welcome to the documentation page of binary_tree!

CHAPTER 1

Contents

1.1 About

binary_tree provides a *Node* object, *node* functions, and *tree* functions for a binary tree data structure.

1.1.1 Installation

To install *binary_tree*, run this in your terminal:

```
$ pip install git+git://github.com/han-keong/binary_tree
```

The conventional way of importing from *binary_tree* is to do:

```
from binary_tree import Node, node, tree
```

You may also import everything by doing:

```
from binary_tree import *
```

1.1.2 Features

Construct a node

Node attributes

Every `Node` has the following attributes:

- Stored value
  - `value`
- Children nodes
  - `left`

  - *right*

- Neighbour nodes
  - *prev*
  - *next*
- Parent node
  - *parent*

Note: All the attributes above besides *value* should be instances of *Node* if they are present.

Node initialization

When initializing a *Node*, a *value* must be provided.

```
>>> left_node = Node(2)
```

Meanwhile, the other attributes can be set using keyword arguments.

```
>>> parent_node = Node(1, left=left_node)
```

Setting Node attributes

Attributes that are reciprocative are set automatically.

For example, when you set the *left* or *right* attribute of a *Node* instance, the child's *parent* attribute is also set behind the scenes.

```
>>> left_node.parent is parent_node
True
```

```
>>> right_node = Node(3)
>>> parent_node.right = right_node
>>>
>>> right_node.parent is parent_node
True
```

Likewise, setting the *prev* or *next* attribute of a *Node* instance will affect the other corresponding neighbour attribute.

```
>>> right_node.prev = left_node
>>>
>>> left_node.next is right_node
True
```

Check a node

The following *node* functions can be used to check if a *Node* has certain properties.

is_node()

is_node() checks if an object is an instance of *Node*.

```
>>> node.is_node(parent_node)
True
```

is_left()

is_left() checks if an instance of *Node* is a left child.

```
>>> node.is_left(parent_node.left)
True
```

is_right()

is_right() checks if an instance of *Node* is a right child.

```
>>> node.is_right(parent_node.right)
True
```

is_leaf()

is_leaf() checks if an instance of *Node* is a leaf node.

```
>>> node.is_leaf(parent_node.right)
True
```

is_root()

is_root() checks if an instance of *Node* is a root node.

```
>>> node.is_root(parent_node):
True
```

is_orphan()

is_orphan() checks if an instance of *Node* is an orphan node.

```
>>> lonely_node = Node(1)
>>> node.is_orphan(lonely_node)
True
```

Equality tests

Node instances have a special way of testing *equality*, which is to tentatively compare the *value* of `self` and the other object.

If the other object does not have a *value* attribute, the object itself is taken as the basis of comparison.

This allows the following comparisons to work:

```
>>> parent_node == Node(1)
True
```

```
>>> parent_node == 1
True
```

If you would like to test if two instances of *Node* have the same binary tree structure, you may compare their *repr()* strings.

```
>>> parent_node2 = Node(1, left=Node(2), right=Node(3))
>>>
>>> repr(parent_node) == repr(parent_node2)
True
```

Set up a binary tree

The *tree* module contains all the relevant functions for binary tree structures.

from_string()

A tree string should be in level-order and separated by commas.

```
>>> tree_string = "1,2,3,4,5,6"
```

Empty spaces can be represented by an immediate comma or `"null"` to be explicit.

```
>>> tree_string = "1,2,3,4,,5,6"
>>> tree_string = "1,2,3,4,null,5,6"
```

Pass the string into *from_string()* to generate a *Node* instance with the desired binary tree structure.

```
>>> root = tree.from_string(tree_string)
```

You can use *repr()* to see the binary tree structure of the *Node* instance.

```
>>> repr(root)
"Node(1, left=Node(2, left=Node(4)), right=Node(3, left=Node(5), right=Node(6)))"
```

from_orders()

Another way to set up a binary tree structure is with its in-order and pre-order traversals.

```
>>> in_order = [4,2,1,5,3,6]
>>> pre_order = [1,2,4,3,5,6]
```

Pass the appropriate key and the traversals into *from_orders()* to generate a *Node* instance with the original tree structure.

```
>>> root = tree.from_orders("in-pre", in_order, pre_order)
>>> repr(root)
"Node(1, left=Node(2, left=Node(4)), right=Node(3, left=Node(5), right=Node(6)))"
```

Alternatively, you can use the in-order and post-order traversal.

```
>>> post_order = [4,2,5,6,3,1]
>>> root = tree.from_orders("in-post", in_order, post_order)
>>>
>>> repr(root)
"Node(1, left=Node(2, left=Node(4)), right=Node(3, left=Node(5), right=Node(6)))"
```

Note: There should not be duplicates present in *in_order* and *pre_order* or *post_order*.

connect_nodes()

When using the above methods to construct a *Node* instance, the neighbour nodes in each level of its binary tree structure are already connected using *connect_nodes()*.

You may use this function again to reconfigure the tree structure of a root *Node* instance after modifying it, or to connect one that was manually set up.

```
>>> root.right.right = None  # Prune the right branch of the right child
>>> tree.connect_nodes(root)
```

to_string()

Just as a binary tree structure can be constructed from string, it can be deconstructed back into one too, using *to_string()*.

```
>>> tree.to_string(root)
"1,2,3,4,,5"
```

Traverse a binary tree

With a binary tree structure set up, there are several *tree* functions you can use to traverse it.

traverse_pre_order()

traverse_pre_order() traverses the binary tree structure of a root *Node* instance in pre-order.

```
>>> list(tree.traverse_pre_order(root))
[Node(1), Node(2), Node(4), Node(3), Node(5)]
```

traverse_in_order()

traverse_in_order() traverses the binary tree structure of a root *Node* instance in in-order.

```
>>> list(tree.traverse_in_order(root))
[Node(4), Node(2), Node(1), Node(5), Node(3)]
```

traverse_post_order()

traverse_post_order() traverses the binary tree structure of a root *Node* instance in post-order.

```
>>> list(tree.traverse_post_order(root))
[Node(4), Node(2), Node(5), Node(3), Node(1)]
```

traverse_level_order()

traverse_level_order() traverses the binary tree structure of a root *Node* instance in level-order.

```
>>> list(tree.traverse_level_order(root))
[[Node(1)], [Node(2), Node(3)], [Node(4), Node(5)]]
```

Note: *traverse_level_order()* will yield lists containing instances of *Node*. Each list represents a level in the binary tree structure.

---

traverse()

A single dispatch function, *traverse()*, is available for convenience.

```
>>> list(tree.traverse(root, "pre"))
[Node(1), Node(2), Node(4), Node(3), Node(5)]
```

```
>>> list(tree.traverse(root, "in"))
[Node(4), Node(2), Node(1), Node(5), Node(3)]
```

```
>>> list(tree.traverse(root, "post"))
[Node(4), Node(2), Node(5), Node(3), Node(1)]
```

```
>>> list(tree.traverse(root, "level"))
[[Node(1)], [Node(2), Node(3)], [Node(4), Node(5)]]
```

Iterating over a Node

You can also *iterate* over an instance of *Node* to traverse its binary tree structure.

```
>>> for node in root:
...     print(node)
Node(1)
Node(2)
Node(3)
Node(4)
Node(5)
```

Note: Iteration over a *Node* instance goes by level-order traversal.

Analyze a binary tree

The following *tree* functions are available to find certain properties of a binary tree structure.

is_symmetrical()

is_symmetrical() checks for symmetry in the binary tree structure of a root *Node* instance.

```
>>> tree.is_symmetrical(root)
False
```

max_depth()

max_depth() calculates the maximum depth of the binary tree structure of a root *Node* instance.

```
>>> tree.max_depth(root)
3
```

get_path()

get_path() traces the ancestry of a *Node* instance.

```
>>> tree.get_path(root.right.left)
[Node(1), Node(3), Node(5)]
```

all_paths()

all_paths() finds every leaf path in the binary tree structure of a root *Node* instance.

```
>>> for path in tree.all_paths(root):
...     print(path)
[Node(1), Node(2), Node(4)]
[Node(1), Node(3), Node(5)]
```

Note: *all_paths()* searches for paths using post-order traversal.

has_sum()

has_sum() determines if there is a path in the binary tree structure of a root *Node* instance that adds up to a certain value.

```
>>> tree.has_sum(root, 7)
True
```

find_path()

`find_path()` finds the path of some `Node` instance or value in the binary tree structure of a root `Node` instance.

```
>>> tree.find_path(5)
[Node(1), Node(3), Node(5)]
```

```
>>> tree.find_path(2)
[Node(1), Node(2)]
```

get_lca()

`get_lca()` gets the lowest common ancestor of two or more `Node` instances or values in the binary tree structure of a root `Node` instance.

```
>>> tree.get_lca(root, 2, 4)
Node(2)
```

```
>>> tree.get_lca(root, 1, 3, 5)
Node(1)
```

Note: It is possible to pass the value of the `Node` instance you wish to refer to because of *the way equality is tested for*. However, the value *must be unique* within the binary tree structure.

1.1.3 Credits

`binary_tree` was written by Han Keong <hk997@live.com>.

This package was created with Cookiecutter and the audreyr/cookiecutter-pypackage project template.

1.2 Documentation

This module provides a Node class, node functions, and tree functions for a binary tree data structure.

Example

```
from binary_tree import from_string, from_orders, traverse

node = from_string("1,2,,3,4,,5")

in_order = list(traverse(node, "in"))
pre_order = list(traverse(node, "pre"))
node2 = from_orders("in-pre", in_order, pre_order)
```

```
>>> repr(node) == repr(node2)
True
```

1.2.1 Node

class binary_tree.**Node**(*value*, ***nodes*)

The basic unit of a binary tree structure.

value
The node value.

left
The left child *Node* instance, if present.

right
The right child *Node* instance, if present.

prev
The left neighbouring *Node* instance, if present.

next
The right neighbouring *Node* instance, if present.

parent
The parent *Node* instance, if present.

Comparing the value of a Node instance

Node.**__eq__**(*other*)

Tentatively compare the *value* of `self` and *other*.

If *other* does not have a *value*, use *other* itself as a basis of comparison.

Parameters **other** – Any object.

Returns `True` if the *value* of `self` is equal to the *value* of *other*, or *other* itself- and `False` otherwise.

Getting the binary tree structure of a Node instance

Node.**__repr__**()

Get the full representation of `self`.

repr() comprises of *value*, the *repr()* of *left* if present, and the *repr()* of *right* if present.

Returns A full representation of `self`.

Return type str

Iterating over the binary tree structure of a Node instance

Node.**__iter__**()

Traverse the binary tree structure of `self` in level-order.

Yields A *Node* in the binary tree structure of `self`.

1.2.2 node

This module contains functions for the Node class.

Checking for a Node instance

binary_tree.node.**is_node**(*obj*)
Check if *obj* is an instance of *Node*.

Parameters **obj** – Any object.

Returns `True` if *obj* is an instance of *Node*, `False` otherwise.

Checking for a child Node instance

binary_tree.node.**is_left**(*node*)
Check if *node* is a *left* child.

Returns `True` if *node* is the *left* node of its *parent*, `False` otherwise, or if its *parent* is not set.

binary_tree.node.**is_right**(*node*)
Check if *node* is a *right* child.

Returns `True` if *node* is the *right* node of its *parent*, `False` otherwise, or if its *parent* is not set.

Checking for a Node instance in a binary tree structure

binary_tree.node.**is_leaf**(*node*)
Check if *node* is a leaf node.

Returns `True` if *node* has a *parent* but no *left* or *right* node, `False` otherwise.

binary_tree.node.**is_root**(*node*)
Check if *node* is a root node.

Returns `True` if *node* has a *left* or *right* node but no *parent* node, `False` otherwise.

binary_tree.node.**is_orphan**(*node*)
Check if *node* is an orphan node.

Returns `True` if *node* has no *parent*, *left*, and *right* node, `False` otherwise.

1.2.3 tree

This module contains functions for binary trees.

Constructing a Node instance with a binary tree structure

binary_tree.tree.**from_string**(*tree_string*, *cls=<class 'binary_tree.node.Node'>*)
Construct a *Node* instance with the binary tree structure represented by *tree_string*.

Initializes the root *Node* instance (the first level), followed by *left* and then *right* for every *Node* instance per level (level-order).

Parameters

- **tree_string** (*str*) – A level-order binary tree traversal, separated by commas.
- **cls** (*type*) – The class constructor to use. Defaults to *Node*.

Returns A newly initialized *cls* instance with the binary tree structure that represents *tree_string*. If *tree_string* has no root value, returns `None`.

Note: Empty spaces can be represented by an immediate comma or `"null"` for explicitness.

`binary_tree.tree.`**from_orders**(*kind*, *in_order*, *other_order*, *cls=<class 'binary_tree.node.Node'>*)

Construct a *Node* instance with the binary tree structure that entails *in-order* and *other_order*.

Recursively initializes *parent*, *left*, and then *right*. (pre-order).

Parameters

- **kind** (*str*) – Either "in-pre" or "in-post".
- **in_order** (*list[int, ..]*) – The in-order traversal of a binary tree.
- **other_order** (*list[int, ..]*) – Either the tree's pre-order or post-order traversal.
- **cls** (*type*) – The class constructor to use. Defaults to *Node*.

Returns A newly initialized *cls* instance with the binary tree structure that entails *in_order* and *other_order*. If either arguments are empty, returns `None`.

Raises

- `ValueError` – If *in_order* and *other_order* do not correspond to a binary tree structure or contain duplicates.
- `KeyError` – If *kind* is not one of the accepted keys.

Note: There cannot be any duplicates in *in_order* and *other_order*.

`binary_tree.tree.`**connect_nodes**(*root*)

Connect the *Node* instances in each level of *root*.

Parameters **root** – A root *Node* instance.

`binary_tree.tree.`**to_string**(*root*)

Deconstruct *root* into a string.

Parameters **root** – A root *Node* instance.

Returns A level-order binary tree traversal, separated by commas.

Return type str

Note: Empty spaces in the tree string are indicated with `"null"`.

Traversing a Node instance with a binary tree structure

`binary_tree.tree.`**traverse_pre_order**(*root*)

Traverse *root* in pre-order.

Visit *parent*, *left*, and then *right*.

Parameters **root** – A root *Node* instance.

Yields A *Node* instance in the binary tree structure of *root*.

binary_tree.tree.**traverse_in_order**(*root*)
Traverse *root* in in-order.

Visit *left*, *parent*, and then *right*.

Parameters **root** – A root *Node* instance.

Yields A *Node* instance in the binary tree structure of *root*.

binary_tree.tree.**traverse_post_order**(*root*)
Traverse *root* in post-order.

Visit *left*, *right*, and then *parent*.

Parameters **root** – A root *Node* instance.

Yields A *Node* instance in the binary tree structure of *root*.

binary_tree.tree.**traverse_level_order**(*root*)
Traverse *root* in level-order.

Visit *root* (the first level), followed by *left* and then *right* for every *Node* instance per level.

Parameters **root** – A root *Node* instance.

Yields A list of *Node* instances representing a level in *root*.

binary_tree.tree.**traverse**(*root*, *kind*)
Forward *root* to the *kind* of traversal.

Parameters

- **root** – A root *Node* instance.
- **kind** (*str*) – "pre" or "in" or "post" or "level".

Returns The generator iterator of the *kind* of traversal (with *root* passed to it).

Raises KeyError – If *kind* is not one of the possible options.

Analyzing a Node instance with a binary tree structure

binary_tree.tree.**is_symmetrical**(*root*)
Check for symmetry in *root*.

Parameters **root** – A root *Node* instance.

Returns True if the binary tree structure of *root* is symmetrical, False otherwise.

binary_tree.tree.**max_depth**(*root*)
Calculate the maximum depth of *root*.

Parameters **root** – A root *Node* instance.

Returns The total number of levels in the binary tree structure of *root*.

Return type int

binary_tree.tree.**get_path**(*node*)
Trace the ancestry of *node*.

Parameters **node** – A *Node* instance in a binary tree.

Returns A list of *Node* instances from the greatest ancestor to *node*.

binary_tree.tree.**all_paths**(*root*)
Find every leaf path in *root*.

Search for leaf nodes in *root* using post-order traversal.

Parameters **root** – A root *Node* instance.

Yields A list of *Node* instances from *root* to a leaf *Node* instance.

binary_tree.tree.**has_sum**(*root*, *value*)
Determine if there is a path in *root* that adds up to *value*.

Parameters

- **root** – A root *Node* instance.
- **value** – The sum to check for.

Returns `True` if a path that adds up to *value* exists in *root*, `False` otherwise.

binary_tree.tree.**find_path**(*root*, *node*)
Find the path of (the *Node* instance of) *node* in *root*.

Parameters

- **root** – A root *Node* instance.
- **node** – A *Node* instance or value in *root*.

Returns A list of every *Node* instance from *root* to (the *Node* instance of) *node*, or `None` if *node* is absent in *root*.

Note: If *node* is a value, it must be unique within the binary tree structure of *root*.

---

binary_tree.tree.**get_lca**(*root*, **nodes*)
Get the lowest common ancestor of two or more (*Node* instances of) *nodes* in *root*.

Parameters

- **root** – A root *Node* instance.
- ***nodes** (Node) – *Node* instances or values in *root*.

Returns The *Node* instance that is the lowest common ancestor of (the *Node* instances of) *nodes* in *root*, or `None` if there is no common ancestor.

Note: Values in *nodes* must be unique within the binary tree structure of *root*.

Python Module Index

b

Index

Symbols

A

B

C

F

G

H

I

L

M

N

P

R

T

V