binarytree Documentation

Release 3.0.1

Joohwan Oh

May 11, 2018

Contents

Welcome to the documentation for **binarytree**!

Binarytree is a Python library which provides a simple API to generate, visualize, inspect and manipulate binary trees. It allows you to skip the tedious work of setting up test data, and dive straight into practising your algorithms! Heaps and BSTs (binary search trees) are also supported.

CHAPTER 1

Requirements

- Python 2.7, 3.4, 3.5 or 3.6
- Pip installer

CHAPTER 2

Installation

To install a stable version from PyPi:

```
~$ pip install binarytree
```

To install the latest version directly from GitHub:

```
~$ pip install -e git+git@github.com:joowani/binarytree.git@master#egg=binarytree
```

You may need to use `sudo` depending on your environment.

CHAPTER 3

Contents

3.1 Overview

By default, **binarytree** uses the following class to represent a node:

```
class Node(object):

    def __init__(self, value, left=None, right=None):
        self.value = value  # The node value
        self.left = left    # Left child
        self.right = right  # Right child
```

Generate and pretty-print various types of binary trees:

```
>>> from binarytree import tree, bst, heap
>>>
>>> # Generate a random binary tree and return its root node
>>> my_tree = tree(height=3, is_perfect=False)
>>>
>>> # Generate a random BST and return its root node
>>> my_bst = bst(height=3, is_perfect=True)
>>>
>>> # Generate a random max heap and return its root node
>>> my_heap = heap(height=3, is_max=True, is_perfect=False)
>>>
>>> # Pretty-print the trees in stdout
>>> print(my_tree)

    ________1_____
   /              \
  4__           ___3
 /   \         /    \
0     9       13     14
     / \        \
```

(continues on next page)

(continued from previous page)

```
    7   10      2

>>> print(my_bst)

        _______7________
       /                \
    __3__            ___11___
   /     \          /         \
  1       5        9          _13
 / \     / \      / \        /    \
0   2   4   6    8   10     12     14

>>> print(my_heap)

          ______14__
         /          \
    _____13__         9
   /         \       / \
  12          7     3   8
 /  \        /
0    10     6
```

Use the *binarytree.Node* class to build your own trees:

```
>>> from binarytree import Node
>>>
>>> root = Node(1)
>>> root.left = Node(2)
>>> root.right = Node(3)
>>> root.left.right = Node(4)
>>>
>>> print(root)

  __1
 /   \
2     3
 \
  4
```

Inspect tree properties:

```
>>> from binarytree import Node
>>>
>>> root = Node(1)
>>> root.left = Node(2)
>>> root.right = Node(3)
>>> root.left.left = Node(4)
>>> root.left.right = Node(5)
>>>
>>> print(root)

    __1
   /   \
  2     3
 / \
4   5
```

(continues on next page)

(continued from previous page)

```
>>> root.height
2
>>> root.is_balanced
True
>>> root.is_bst
False
>>> root.is_complete
True
>>> root.is_max_heap
False
>>> root.is_min_heap
True
>>> root.is_perfect
False
>>> root.is_strict
True
>>> root.leaf_count
3
>>> root.max_leaf_depth
2
>>> root.max_node_value
5
>>> root.min_leaf_depth
1
>>> root.min_node_value
1
>>> root.size
5

>>> root.properties
{'height': 2,
 'is_balanced': True,
 'is_bst': False,
 'is_complete': True,
 'is_max_heap': False,
 'is_min_heap': True,
 'is_perfect': False,
 'is_strict': True,
 'leaf_count': 3,
 'max_leaf_depth': 2,
 'max_node_value': 5,
 'min_leaf_depth': 1,
 'min_node_value': 1,
 'size': 5}

>>> root.leaves
[Node(3), Node(4), Node(5)]

>>> root.levels
[[Node(1)], [Node(2), Node(3)], [Node(4), Node(5)]]
```

Use level-order (breath-first) indexes to manipulate nodes:

```
>>> from binarytree import Node
>>>
>>> root = Node(1)                    # index: 0, value: 1
>>> root.left = Node(2)               # index: 1, value: 2
```

(continues on next page)

(continued from previous page)

```
>>> root.right = Node(3)             # index: 2, value: 3
>>> root.left.right = Node(4)        # index: 4, value: 4
>>> root.left.right.left = Node(5)   # index: 9, value: 5
>>>
>>> print(root)

  ____1
 /     \
2__     3
   \
    4
   /
  5

>>> # Use binarytree.Node.pprint instead of print to display indexes
>>> root.pprint(index=True)

   __________0-1_
  /              \
1-2_____         2-3
        \
       _4-4
      /
    9-5

>>> # Return the node/subtree at index 9
>>> root[9]
Node(5)

>>> # Replace the node/subtree at index 4
>>> root[4] = Node(6, left=Node(7), right=Node(8))
>>> root.pprint(index=True)

   _________________0-1_
  /                     \
1-2_____                2-3
        \
       _4-6_
      /     \
    9-7     10-8

>>> # Delete the node/subtree at index 1
>>> del root[1]
>>> root.pprint(index=True)

0-1_
    \
    2-3
```

Traverse the trees using different algorithms:

```
>>> from binarytree import Node
>>>
>>> root = Node(1)
>>> root.left = Node(2)
>>> root.right = Node(3)
>>> root.left.left = Node(4)
```

(continues on next page)

(continued from previous page)

```
>>> root.left.right = Node(5)
>>>
>>> print(root)

    __1
   /   \
  2     3
 / \
4   5

>>> root.inorder
[Node(4), Node(2), Node(5), Node(1), Node(3)]

>>> root.preorder
[Node(1), Node(2), Node(4), Node(5), Node(3)]

>>> root.postorder
[Node(4), Node(5), Node(2), Node(3), Node(1)]

>>> root.levelorder
[Node(1), Node(2), Node(3), Node(4), Node(5)]
```

List representations are also supported:

```
>>> from binarytree import build
>>>
>>> # Build a tree from list representation
>>> root = build([7, 3, 2, 6, 9, None, 1, 5, 8])
>>> print(root)

        __7
       /   \
    __3     2
   /   \     \
  6     9     1
 / \
5   8

>>> # Convert the tree back to list representation
>>> list(root)
[7, 3, 2, 6, 9, None, 1, 5, 8]
```

See *API Specification* for more details.

3.2 API Specification

This page contains the API specification for the *binarytree.Node* class, and utility functions *binarytree.build*, *binarytree.tree*, *binarytree.bst* and *binarytree.heap*.

3.2.1 Class: binarytree.Node

class binarytree.**Node**(*value*, *left=None*, *right=None*)

Represents a binary tree node.

This class provides methods and properties for managing the calling node, and the binary tree which the calling node is the root of. Whenever a docstring in this class says "binary tree", it is referring to the calling node instance and its descendants.

Parameters

- **value** (*int*) – The node value. Only integers are supported.
- **left** (*binarytree.Node | None*) – The left child node (default: None).
- **right** (*binarytree.Node | None*) – The right child node (default: None).

Raises

- ***binarytree.exceptions.InvalidNodeValueError*** – If the node value is not an integer.
- ***binarytree.exceptions.InvalidNodeTypeError*** – If the left or right child is not an instance of *binarytree.Node*.

__delitem__(*index*)

Remove the node/subtree at the given level-order (breath-first) index from the binary tree.

- An exception is raised if the target node does not exist.
- The descendants of the target node (if any) are also removed.
- The root node (calling node) cannot be deleted.

Parameters **index** (*int*) – The node index.

Raises

- ***binarytree.exceptions.OperationForbiddenError*** – If the user tries to delete the root node (calling node).
- ***binarytree.exceptions.NodeNotFoundError*** – If the target node or its parent does not exist.

Example:

```
>>> from binarytree import Node
>>>
>>> root = Node(1)           # index: 0, value: 1
>>> root.left = Node(2)      # index: 1, value: 2
>>> root.right = Node(3)     # index: 2, value: 3
>>>
>>> del root[0]
Traceback (most recent call last):
  ...
OperationForbiddenError: Cannot delete the root node
```

```
>>> from binarytree import Node
>>>
>>> root = Node(1)           # index: 0, value: 1
>>> root.left = Node(2)      # index: 1, value: 2
>>> root.right = Node(3)     # index: 2, value: 3
>>>
>>> del root[2]
>>>
>>> root[2]
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
 ...
NodeNotFoundError: Node missing at index 2
```

__getitem__(*index*)

Return the node/subtree at the give level-order (breath-first) index.

Parameters **index** (*int*) – The node index.

Returns The node at the given index.

Return type *binarytree.Node*

Raises

- ***binarytree.exceptions.InvalidNodeIndexError*** – If an invalid index is given.
- ***binarytree.exceptions.NodeNotFoundError*** – If the target node does not exist.

Example:

```
>>> from binarytree import Node
>>>
>>> root = Node(1)        # index: 0, value: 1
>>> root.left = Node(2)   # index: 1, value: 2
>>> root.right = Node(3) # index: 2, value: 3
>>>
>>> root[0]
Node(1)
>>> root[1]
Node(2)
>>> root[2]
Node(3)
>>> root[3]
Traceback (most recent call last):
 ...
NodeNotFoundError: Node missing at index 3
```

__init__(*value*, *left=None*, *right=None*)

x.__init__(...) initializes x; see help(type(x)) for signature

__iter__()

Return the list representation of the binary tree.

Returns The list representation consisting of node values or None's. If a node has an index i, its left child is at index 2i + 1, right child at index 2i + 2, and parent at index floor((i - 1) / 2). None signifies the absence of a node. See example below for an illustration.

Return type [int | None]

Example:

```
>>> from binarytree import Node
>>>
>>> root = Node(1)
>>> root.left = Node(2)
>>> root.right = Node(3)
>>> root.left.right = Node(4)
>>>
```

(continues on next page)

(continued from previous page)

```
>>> list(root)
[1, 2, 3, None, 4]
```

__len__()

Return the total number of nodes in the binary tree.

Returns The total number of nodes.

Return type int

Example:

```
>>> from binarytree import Node
>>>
>>> root = Node(1)
>>> root.left = Node(2)
>>> root.right = Node(3)
>>>
>>> len(root)
3
```

Note: This method is equivalent to *binarytree.Node.size*.

__setitem__(*index*, *node*)

Insert the node/subtree into the binary tree at the given level-order (breath-first) index.

- An exception is raised if the parent node does not exist.
- Any existing node/subtree is overwritten.
- The root node (calling node) cannot be replaced.

Parameters

- **index** (*int*) – The node index.
- **node** (binarytree.Node) – The new node to insert.

Raises

- ***binarytree.exceptions.OperationForbiddenError*** – If the user tries to overwrite the root node (calling node).
- ***binarytree.exceptions.NodeNotFoundError*** – If the parent for the new node does not exist.
- ***binarytree.exceptions.InvalidNodeTypeError*** – If the new node is not an instance of *binarytree.Node*.

Example:

```
>>> from binarytree import Node
>>>
>>> root = Node(1)        # index: 0, value: 1
>>> root.left = Node(2)   # index: 1, value: 2
>>> root.right = Node(3)  # index: 2, value: 3
>>>
>>> root[0] = Node(4)
```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
  ...
OperationForbiddenError: Cannot modify the root node
```

```
>>> from binarytree import Node
>>>
>>> root = Node(1)        # index: 0, value: 1
>>> root.left = Node(2)   # index: 1, value: 2
>>> root.right = Node(3)  # index: 2, value: 3
>>>
>>> root[11] = Node(4)
Traceback (most recent call last):
  ...
NodeNotFoundError: Parent node missing at index 5
```

```
>>> from binarytree import Node
>>>
>>> root = Node(1)        # index: 0, value: 1
>>> root.left = Node(2)   # index: 1, value: 2
>>> root.right = Node(3)  # index: 2, value: 3
>>>
>>> root[1] = Node(4)
>>>
>>> root.left
Node(4)
```

__str__()

Return the pretty-print string for the binary tree.

Returns The pretty-print string.

Return type str | unicode

Example:

```
>>> from binarytree import Node
>>>
>>> root = Node(1)
>>> root.left = Node(2)
>>> root.right = Node(3)
>>> root.left.right = Node(4)
>>>
>>> print(root)

  __1
 /   \
2     3
 \
  4
```

Note: To include level-order (breath-first) indexes in the string, use *binarytree.Node.pprint()* instead.

height

Return the height of the binary tree.

Returns The height of the binary tree.

Return type int

Example:

```
>>> from binarytree import Node
>>>
>>> root = Node(1)
>>> root.left = Node(2)
>>> root.left.left = Node(3)
>>>
>>> print(root)

    1
   /
  2
 /
3

>>> root.height
2
```

Note: A binary tree with only a root node has a height of 0.

inorder

Return the nodes in the binary tree using in-order (left, root, right) traversal.

Returns The list of nodes.

Return type [*binarytree.Node*]

Example:

```
>>> from binarytree import Node
>>>
>>> root = Node(1)
>>> root.left = Node(2)
>>> root.right = Node(3)
>>> root.left.left = Node(4)
>>> root.left.right = Node(5)
>>>
>>> print(root)

    __1
   /   \
  2     3
 / \
4   5

>>> root.inorder
[Node(4), Node(2), Node(5), Node(1), Node(3)]
```

is_balanced

Return True if the binary tree is height-balanced, False otherwise.

Returns True if the binary tree is balanced, False otherwise.

Return type bool

Example:

```
>>> from binarytree import Node
>>>
>>> root = Node(1)
>>> root.left = Node(2)
>>> root.left.left = Node(3)
>>>
>>> print(root)

    1
   /
  2
 /
3

>>> root.is_balanced
False
```

is_bst

Return True if the binary tree is a BST (binary search tree), False otherwise.

Returns True if the binary tree is a BST, False otherwise.

Return type bool

Example:

```
>>> from binarytree import Node
>>>
>>> root = Node(2)
>>> root.left = Node(1)
>>> root.right = Node(3)
>>>
>>> print(root)

  2
 / \
1   3

>>> root.is_bst
True
```

is_complete

Return True if the binary tree is complete (i.e. all levels except possibly the last are completely filled, and the last level is always left-justified), False otherwise.

Returns True if the binary tree is complete, False otherwise.

Return type bool

Example:

```
>>> from binarytree import Node
>>>
>>> root = Node(1)
>>> root.left = Node(2)
>>> root.right = Node(3)
>>> root.left.left = Node(4)
>>> root.left.right = Node(5)
```

(continues on next page)

(continued from previous page)

```
>>>
>>> print(root)

    __1
   /   \
  2     3
 / \
4   5

>>> root.is_complete
True
```

is_max_heap

Return True if the binary tree is a max heap, False otherwise.

Returns True if the binary tree is a max heap, False otherwise.

Return type bool

Example:

```
>>> from binarytree import Node
>>>
>>> root = Node(3)
>>> root.left = Node(1)
>>> root.right = Node(2)
>>>
>>> print(root)

  3
 / \
1   2

>>> root.is_max_heap
True
```

is_min_heap

Return True if the binary tree is a min heap, False otherwise.

Returns True if the binary tree is a min heap, False otherwise.

Return type bool

Example:

```
>>> from binarytree import Node
>>>
>>> root = Node(1)
>>> root.left = Node(2)
>>> root.right = Node(3)
>>>
>>> print(root)

  1
 / \
2   3

>>> root.is_min_heap
True
```

is_perfect

Return True if the binary tree is perfect (i.e. all levels are completely filled), False otherwise.

Returns True if the binary tree is perfect, False otherwise.

Return type bool

Example:

```
>>> from binarytree import Node
>>>
>>> root = Node(1)
>>> root.left = Node(2)
>>> root.right = Node(3)
>>> root.left.left = Node(4)
>>> root.left.right = Node(5)
>>> root.right.left = Node(6)
>>> root.right.right = Node(7)
>>>
>>> print(root)

    __1__
   /     \
  2       3
 / \     / \
4   5   6   7

>>> root.is_perfect
True
```

is_strict

Return True if the binary tree is strict (i.e. all non-leaf nodes have both children), False otherwise.

Returns True if the binary tree is strict, False otherwise.

Return type bool

Example:

```
>>> from binarytree import Node
>>>
>>> root = Node(1)
>>> root.left = Node(2)
>>> root.right = Node(3)
>>> root.left.left = Node(4)
>>> root.left.right = Node(5)
>>>
>>> print(root)

    __1
   /   \
  2     3
 / \
4   5

>>> root.is_strict
True
```

Note: Strictly binary nodes are also called **full** nodes.

leaf_count
Return the total number of leaves in the binary tree.

Returns The total number of leaves.

Return type int

Example:

```
>>> from binarytree import Node
>>>
>>> root = Node(1)
>>> root.left = Node(2)
>>> root.right = Node(3)
>>> root.left.right = Node(4)
>>>
>>> root.leaf_count
2
```

leaves
Return the leaves of the binary tree.

Returns The list of leaf nodes.

Return type [*binarytree.Node*]

Example:

```
>>> from binarytree import Node
>>>
>>> root = Node(1)
>>> root.left = Node(2)
>>> root.right = Node(3)
>>> root.left.right = Node(4)
>>>
>>> print(root)

  __1
 /   \
2     3
 \
  4

>>> root.leaves
[Node(3), Node(4)]
```

levelorder
Return the nodes in the binary tree using level-order (breath-first) traversal.

Returns The list of nodes.

Return type [*binarytree.Node*]

Example:

```
>>> from binarytree import Node
>>>
>>> root = Node(1)
>>> root.left = Node(2)
>>> root.right = Node(3)
>>> root.left.left = Node(4)
```

(continues on next page)

(continued from previous page)

```
>>> root.left.right = Node(5)
>>>
>>> print(root)

    __1
   /   \
  2     3
 / \
4   5

>>> root.levelorder
[Node(1), Node(2), Node(3), Node(4), Node(5)]
```

levels

Return the nodes in the binary tree level by level.

Returns The per-level lists of nodes.

Return type [[*binarytree.Node*]]

Example:

```
>>> from binarytree import Node
>>>
>>> root = Node(1)
>>> root.left = Node(2)
>>> root.right = Node(3)
>>> root.left.right = Node(4)
>>>
>>> print(root)

  __1
 /   \
2     3
 \
  4

>>>
>>> root.levels
[[Node(1)], [Node(2), Node(3)], [Node(4)]]
```

max_leaf_depth

Return the maximum leaf node depth in the binary tree.

Returns The maximum leaf node depth.

Return type int

Example:

```
>>> from binarytree import Node
>>>
>>> root = Node(1)
>>> root.left = Node(2)
>>> root.right = Node(3)
>>> root.right.left = Node(4)
>>> root.right.left.left = Node(5)
>>>
```

(continues on next page)

(continued from previous page)

```
>>> print(root)

  1_____
 /      \
2        3
        /
       4
      /
     5

>>> root.max_leaf_depth
3
```

max_node_value

Return the maximum node value in the binary tree.

Returns The maximum node value.

Return type int

Example:

```
>>> from binarytree import Node
>>>
>>> root = Node(1)
>>> root.left = Node(2)
>>> root.right = Node(3)
>>>
>>> root.max_node_value
3
```

min_leaf_depth

Return the minimum leaf node depth in the binary tree.

Returns The minimum leaf node depth.

Return type int

Example:

```
>>> from binarytree import Node
>>>
>>> root = Node(1)
>>> root.left = Node(2)
>>> root.right = Node(3)
>>> root.right.left = Node(4)
>>> root.right.left.left = Node(5)
>>>
>>> print(root)

  1_____
 /      \
2        3
        /
       4
      /
     5
```

(continues on next page)

(continued from previous page)

```
>>> root.min_leaf_depth
1
```

min_node_value

Return the minimum node value in the binary tree.

Returns The minimum node value.

Return type int

Example:

```
>>> from binarytree import Node
>>>
>>> root = Node(1)
>>> root.left = Node(2)
>>> root.right = Node(3)
>>>
>>> root.min_node_value
1
```

postorder

Return the nodes in the binary tree using post-order (left, right, root) traversal.

Returns The list of nodes.

Return type [*binarytree.Node*]

Example:

```
>>> from binarytree import Node
>>>
>>> root = Node(1)
>>> root.left = Node(2)
>>> root.right = Node(3)
>>> root.left.left = Node(4)
>>> root.left.right = Node(5)
>>>
>>> print(root)

    __1
   /   \
  2     3
 / \
4   5

>>> root.postorder
[Node(4), Node(5), Node(2), Node(3), Node(1)]
```

pprint (*index=False*, *delimiter=u'-'*)

Pretty-print the binary tree.

Parameters

- **index** (*bool*) – If set to True (default: False), display the level-order (breath-first) indexes using the following format: "{index}{delimiter}{value}".
- **delimiter** (*str | unicode*) – The delimiter character between the node index, and the node value (default: "-").

Example:

```
>>> from binarytree import Node
>>>
>>> root = Node(1)               # index: 0, value: 1
>>> root.left = Node(2)          # index: 1, value: 2
>>> root.right = Node(3)         # index: 2, value: 3
>>> root.left.right = Node(4)    # index: 4, value: 4
>>>
>>> root.pprint()

  __1
 /   \
2     3
 \
  4

>>> root.pprint(index=True)       # Format: {index}-{value}

   _____0-1_
  /         \
1-2_        2-3
    \
    4-4
```

Note: If you don't need to see the node indexes, you can use *binarytree.Node.__str__()*.

preorder

Return the nodes in the binary tree using pre-order (root, left, right) traversal.

Returns The list of nodes.

Return type [*binarytree.Node*]

Example:

```
>>> from binarytree import Node
>>>
>>> root = Node(1)
>>> root.left = Node(2)
>>> root.right = Node(3)
>>> root.left.left = Node(4)
>>> root.left.right = Node(5)
>>>
>>> print(root)

    __1
   /   \
  2     3
 / \
4   5

>>> root.preorder
[Node(1), Node(2), Node(4), Node(5), Node(3)]
```

properties

Return various properties of the the binary tree all at once.

Returns The properties of the binary tree.

Return type dict

Example:

```
>>> from binarytree import Node
>>>
>>> root = Node(1)
>>> root.left = Node(2)
>>> root.right = Node(3)
>>> root.left.left = Node(4)
>>> root.left.right = Node(5)
>>> props = root.properties
>>>
>>> props['height']         # equivalent to root.height
2
>>> props['size']           # equivalent to root.size
5
>>> props['max_leaf_depth'] # equivalent to root.max_leaf_depth
2
>>> props['min_leaf_depth'] # equivalent to root.min_leaf_depth
1
>>> props['max_node_value'] # equivalent to root.max_node_value
5
>>> props['min_node_value'] # equivalent to root.min_node_value
1
>>> props['leaf_count']     # equivalent to root.leaf_count
3
>>> props['is_balanced']    # equivalent to root.is_balanced
True
>>> props['is_bst']         # equivalent to root.is_bst
False
>>> props['is_complete']    # equivalent to root.is_complete
True
>>> props['is_max_heap']    # equivalent to root.is_max_heap
False
>>> props['is_min_heap']    # equivalent to root.is_min_heap
True
>>> props['is_perfect']     # equivalent to root.is_perfect
False
>>> props['is_strict']      # equivalent to root.is_strict
True
```

size

Return the total number of nodes in the binary tree.

Returns The total number of nodes.

Return type int

Example:

```
>>> from binarytree import Node
>>>
>>> root = Node(1)
>>> root.left = Node(2)
>>> root.right = Node(3)
>>> root.left.right = Node(4)
>>>
```

(continues on next page)

(continued from previous page)

```
>>> root.size
4
```

Note: This method is equivalent to *binarytree.Node.__len__()*.

validate()
Check if the binary tree is malformed.

Raises

- ***binarytree.exceptions.CyclicNodeReferenceError*** – If there is a cyclic reference to a node in the binary tree.
- ***binarytree.exceptions.InvalidNodeTypeError*** – If a node is not an instance of *binarytree.Node*.
- ***binarytree.exceptions.InvalidNodeValueError*** – If a node value is not an integer.

Example:

```
>>> from binarytree import Node
>>>
>>> root = Node(1)
>>> root.left = Node(2)
>>> root.right = root  # Cyclic reference to root
>>>
>>> root.validate()
Traceback (most recent call last):
 ...
CyclicNodeReferenceError: Cyclic node reference at index 0
```

3.2.2 Function: binarytree.build

binarytree.**build**(*values*)
Build a binary tree from a list representation (i.e. a list of node values and/or None's in breath-first order) and return its root.

Parameters **values** (*[int | None]*) – The list representation (i.e. a list of node values and/or None's in breath-first order). If a node has an index i, its left child is at index 2i + 1, right child at index 2i + 2, and parent at index floor((i - 1) / 2). None signifies the absence of a node. See example below for an illustration.

Returns The root of the binary tree.

Return type *binarytree.Node*

Raises ***binarytree.exceptions.NodeNotFoundError*** – If the list representation is malformed and a parent node is missing.

Example:

```
>>> from binarytree import build
>>>
>>> root = build([1, 2, 3, None, 4])
>>>
```

(continues on next page)

(continued from previous page)

```
>>> print(root)

  __1
 /   \
2     3
 \
  4
```

```
>>> from binarytree import build
>>>
>>> root = build([None, 2, 3])
Traceback (most recent call last):
 ...
NodeNotFoundError: Parent node missing at index 0
```

3.2.3 Function: binarytree.tree

binarytree.**tree**(*height=3*, *is_perfect=False*)

Generate a random binary tree and return its root node.

Parameters

- **height** (*int*) – The height of the tree (default: 3, range: 0 - 9 inclusive).
- **is_perfect** (*bool*) – If set to True (default: False), a perfect binary tree with all levels filled is returned. When set to False, a perfect binary tree may still be generated and returned by chance.

Returns The root node of the generated tree.

Return type *binarytree.Node*

Raises ***binarytree.exceptions.InvalidTreeHeightError*** – If an invalid tree height is given.

Example:

```
>>> from binarytree import tree
>>>
>>> root = tree()
>>>
>>> root.height
3
```

```
>>> from binarytree import tree
>>>
>>> root = tree(height=5, is_perfect=True)
>>>
>>> root.height
5
>>> root.is_perfect
True
```

```
>>> from binarytree import tree
>>>
>>> root = tree(height=20)
```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
 ...
InvalidTreeHeightError: The height must be an integer between 0 - 9
```

3.2.4 Function: binarytree.bst

binarytree.**bst**(*height=3*, *is_perfect=False*)

Generate a random BST (binary search tree) and return its root node.

Parameters

- **height** (*int*) – The height of the BST (default: 3, range: 0 - 9 inclusive).
- **is_perfect** (*bool*) – If set to True (default: False), a perfect BST with all levels filled is returned. When set to False, a perfect BST may still be generated and returned by chance.

Returns The root node of the generated BST.

Return type *binarytree.Node*

Raises ***binarytree.exceptions.InvalidTreeHeightError*** – If an invalid tree height is given.

Example:

```
>>> from binarytree import bst
>>>
>>> root = bst()
>>>
>>> root.height
3
>>> root.is_bst
True
```

```
>>> from binarytree import bst
>>>
>>> root = bst(10)
Traceback (most recent call last):
 ...
InvalidTreeHeightError: The height must be an integer between 0 - 9
```

3.2.5 Function: binarytree.heap

binarytree.**heap**(*height=3*, *is_max=True*, *is_perfect=False*)

Generate a heap and return its root node.

Parameters

- **height** (*int*) – The height of the heap (default: 3, range: 0 - 9 inclusive).
- **is_max** (*bool*) – If set to True (default: True), generate a max heap. Otherwise, generate a min heap. Note that a binary tree with only the root is both a min and max heap.
- **is_perfect** (*bool*) – If set to True (default: False), a perfect heap with all levels filled is returned. When set to False, a perfect heap may still be generated and returned by chance.

Returns The root node of the generated heap.

Return type *binarytree.Node*

Raises ***binarytree.exceptions.InvalidTreeHeightError*** – If an invalid tree height is given.

Example:

```
>>> from binarytree import heap
>>>
>>> root = heap()
>>>
>>> root.height
3
>>> root.is_max_heap
True
```

```
>>> from binarytree import heap
>>>
>>> root = heap(4, is_max=False)
>>>
>>> root.height
4
>>> root.is_min_heap
True
```

```
>>> from binarytree import heap
>>>
>>> root = heap(5, is_max=False, is_perfect=True)
>>>
>>> root.height
5
>>> root.is_min_heap
True
>>> root.is_perfect
True
```

```
>>> from binarytree import heap
>>>
>>> root = heap(-1)
Traceback (most recent call last):
 ...
InvalidTreeHeightError: The height must be an integer between 0 - 9
```

3.3 Exceptions

Below is the list of exceptions raised by **binarytree**:

exception binarytree.exceptions.**BinaryTreeError**
Base exception.

exception binarytree.exceptions.**CyclicNodeReferenceError**
Raised if the binary tree has a cyclic reference to a node.

exception binarytree.exceptions.**InvalidNodeIndexError**
Raised if an invalid level-order index is given.

exception binarytree.exceptions.**InvalidNodeTypeError**
Raised if a node is not an instance of *binarytree.Node*.

exception binarytree.exceptions.**InvalidNodeValueError**
Raised if a node has an invalid value.

exception binarytree.exceptions.**InvalidTreeHeightError**
Raised if an invalid tree height is given.

exception binarytree.exceptions.**NodeNotFoundError**
Raised if a node is missing from the binary tree.

exception binarytree.exceptions.**OperationForbiddenError**
Raised if the user tries to overwrite or delete the root node.

3.4 Contributing

3.4.1 Instructions

Before submitting a pull request on GitHub, please make sure you meet the following requirements:

- The pull request points to the dev (development) branch.
- All changes are squashed into a single commit (I like to use `git rebase -i` to do this).
- The commit message is in present tense (ok: "Add feature", not ok: "Added feature").
- Correct and consistent style: Sphinx-compatible docstrings, using snake vs. camel casing properly and PEP8 compliance (see below).
- No classes/methods/functions with missing docstrings or commented-out lines. You can refer to existing ones for examples.
- The test coverage remains at %100. Sometimes you may find yourself having to write superfluous unit tests to keep this number up. If a piece of code is trivial and has no need for unittests, use this to exclude it from coverage.
- No build failures on TravisCI. The builds automatically trigger on PR submissions.
- Does not break backward-compatibility (unless there is a really good reason).
- Compatible with Python versions 2.7, 3.4, 3.5 and 3.6.

Warning: The dev branch is occasionally rebased, and its commit history may be overwritten in the process (I try very hard never to do this). So before you begin your feature work, git fetch/pull to ensure that branches have not diverged. If you see git conflicts and just want to start from scratch, run this command:

```
~$ git checkout dev
~$ git fetch origin
~$ git reset --hard origin/dev  # THIS WILL WIPE AL LOCAL CHANGES
```

3.4.2 Style

To ensure PEP8 compliance, run flake8:

```
~$ pip install flake8
~$ git clone https://github.com/joowani/binarytree.git
~$ cd binarytree
~$ flake8
```

You should try to resolve all issues reported. If there is a good reason to ignore errors from a specific piece of code, visit here to see how to exclude the lines from the check.

3.4.3 Testing

To test your changes, run the unit tests that come with **binarytree** on your local machine. The tests use pytest.

To run the unit tests:

```
~$ pip install pytest
~$ git clone https://github.com/joowani/binarytree.git
~$ cd binarytree
~$ py.test tests.py --verbose
```

To run the unit tests with coverage report:

```
~$ pip install coverage pytest pytest-cov
~$ git clone https://github.com/joowani/binarytree.git
~$ cd binarytree
~$ py.test tests.py --verbose --cov-report=html --cov=binarytree
~$ # Open the generated file htmlcov/index.html in a browser
```

3.4.4 Documentation

The documentation (including the README) is written in reStructuredText and uses Sphinx. To build the HTML version of the documentation on your local machine:

```
~$ pip install sphinx sphinx_rtd_theme
~$ git clone https://github.com/joowani/binarytree.git
~$ cd binarytree/docs
~$ sphinx-build . build
~$ # Open the generated file build/index.html in a browser
```

As always, thanks for your contribution!

Python Module Index

b

Index

Symbols

B

C

H

I

L

M

N

O

P

S

T

V