binarytree Documentation

Release 4.0.0

Joohwan Oh

Jun 06, 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

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

>>> properties = root.properties  # Get all properties at once
>>> properties['height']
2
>>> properties['is_balanced']
True
>>> properties['max_leaf_depth']
2

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

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

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

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

  ____1
 /     \
```

(continues on next page)

(continued from previous page)

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

    __1
   /   \
  2     3
 / \
```

(continues on next page)

(continued from previous page)

```
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(root)  # Equivalent to 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
>>> values = [7, 3, 2, 6, 9, None, 1, 5, 8]
>>> root = build(values)
>>> print(root)

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

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

See *API Specification* for more details.

3.2 API Specification

This page covers the API specification for the following classes and utility functions:

- *binarytree.Node*
- *binarytree.build()*
- *binarytree.tree()*
- *binarytree.bst()*
- *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 current node instance, and the binary tree in which the node is the root of. When a docstring in this class mentions "binary tree", it is referring to the current node and its descendants.

Parameters

- **value** (*int | float*) – Node value (must be a number).
- **left** (*binarytree.Node | None*) – Left child node (default: None).
- **right** (*binarytree.Node | None*) – Right child node (default: None).

Raises

- ***binarytree.exceptions.NodeTypeError*** – If left or right child node is not an instance of *binarytree.Node*.
- ***binarytree.exceptions.NodeValueError*** – If node value is not a number (e.g. int, float).

__delitem__(*index*)

Remove the node (or subtree) at the given level-order index.

- An exception is raised if the target node is missing.
- The descendants of the target node (if any) are also removed.
- Root node (current node) cannot be deleted.

Parameters **index** (*int*) – Level-order index of the node.

Raises

- ***binarytree.exceptions.NodeNotFoundError*** – If the target node or its parent is missing.
- ***binarytree.exceptions.NodeModifyError*** – If user attempts to delete the root node (current 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
>>>
>>> del root[0]
Traceback (most recent call last):
 ...
NodeModifyError: 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
```

(continues on next page)

(continued from previous page)

```
>>>
>>> del root[2]
>>>
>>> root[2]
Traceback (most recent call last):
 ...
NodeNotFoundError: node missing at index 2
```

`__getitem__`(*index*)

Return the node (or subtree) at the given level-order index.

Parameters **`index`** (*`int`*) – Level-order index of the node.

Returns Node (or subtree) at the given index.

Return type *binarytree.Node*

Raises

- ***`binarytree.exceptions.NodeIndexError`*** – If node index is invalid.
- ***`binarytree.exceptions.NodeNotFoundError`*** – If the node is missing.

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
```

`__iter__`()

Iterate through the nodes in the binary tree in level-order.

Returns Node iterator.

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)
```

(continues on next page)

(continued from previous page)

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

>>> [node for node in root]
[Node(1), Node(2), Node(3), Node(4), Node(5)]
```

__len__()

Return the total number of nodes in the binary tree.

Returns 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 a node (or subtree) at the given level-order index.

- An exception is raised if the parent node is missing.
- Any existing node or subtree is overwritten.
- Root node (current node) cannot be replaced.

Parameters

- **index** (*int*) – Level-order index of the node.
- **node** (binarytree.Node) – Node to insert.

Raises

- ***binarytree.exceptions.NodeTypeError*** – If new node is not an instance of *binarytree.Node*.
- ***binarytree.exceptions.NodeNotFoundError*** – If parent is missing.
- ***binarytree.exceptions.NodeModifyError*** – If user attempts to overwrite the root node (current 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)
Traceback (most recent call last):
 ...
NodeModifyError: 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 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 indexes in the output string, use *binarytree.Node.pprint()* instead.

height

Return the height of the binary tree.

Height of a binary tree is the number of edges on the longest path between the root node and a leaf node. Binary tree with just a single node has a height of 0.

Returns 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 traversal.

An in-order traversal visits left subtree, root, then right subtree.

Returns 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
```

(continues on next page)

(continued from previous page)

```
 / \
4   5

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

is_balanced

Check if the binary tree is height-balanced.

A binary tree is height-balanced if it meets the following criteria:

- Left subtree is height-balanced.
- Right subtree is height-balanced.
- The difference between heights of left and right subtrees is no more than 1.
- An empty binary tree is always height-balanced.

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

Check if the binary tree is a BST (binary search tree).

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)
```

(continues on next page)

(continued from previous page)

```
  2
 / \
1   3

>>> root.is_bst
True
```

is_complete

Check if the binary tree is complete.

A binary tree is complete if it meets the following criteria:

- All levels except possibly the last are completely filled.
- Last level is left-justified.

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)
>>>
>>> print(root)

    __1
   /   \
  2     3
 / \
4   5

>>> root.is_complete
True
```

is_max_heap

Check if the binary tree is a max heap.

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)
```

(continues on next page)

(continued from previous page)

```
  3
 / \
1   2

>>> root.is_max_heap
True
```

is_min_heap

Check if the binary tree is a min heap.

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

Check if the binary tree is perfect.

A binary tree is perfect if all its levels are completely filled. See example below for an illustration.

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
```

(continues on next page)

(continued from previous page)

```
>>> root.is_perfect
True
```

is_strict

Check if the binary tree is strict.

A binary tree is strict if all its non-leaf nodes have both the left and right child nodes.

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
```

leaf_count

Return the total number of leaf nodes in the binary tree.

A leaf node is a node with no child nodes.

Returns Total number of leaf 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)
>>>
>>> root.leaf_count
2
```

leaves

Return the leaf nodes of the binary tree.

A leaf node is any node that does not have child nodes.

Returns 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 traversal.

A level-order traversal visits nodes left to right, level by level.

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

`levels`

Return the nodes in the binary tree level by level.

Returns Lists of nodes level by level.

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 of the binary tree.

Returns 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)
>>>
>>> print(root)

  1____
 /     \
2       3
       /
      4
     /
    5

>>> root.max_leaf_depth
3
```

max_node_value

Return the maximum node value of the binary tree.

Returns 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 of the binary tree.

Returns 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

>>> root.min_leaf_depth
1
```

min_node_value

Return the minimum node value of the binary tree.

Returns 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 traversal.

A post-order traversal visits left subtree, right subtree, then root.

Returns 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 level-order indexes using the format: `{index}{delimiter}{value}`.
- **delimiter** (*str | unicode*) – 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
```

(continues on next page)

(continued from previous page)

```
    \
    4-4
```

Note: If you do not need level-order indexes in the output string, use *binarytree.Node.__str__()* instead.

preorder

Return the nodes in the binary tree using pre-order traversal.

A pre-order traversal visits root, left subtree, then right subtree.

Returns 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 binary tree.

Returns Binary tree properties.

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
```

(continues on next page)

(continued from previous page)

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

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

validate()

Check if the binary tree is malformed.

Raises

- ***binarytree.exceptions.NodeReferenceError*** – If there is a cyclic reference to a node in the binary tree.
- ***binarytree.exceptions.NodeTypeError*** – If a node is not an instance of *binarytree.Node*.

- *binarytree.exceptions.NodeValueError* – If a node value is not a number (e.g. int, float).

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):
 ...
NodeReferenceError: cyclic node reference at index 0
```

values

Return the list representation of the binary tree.

Returns List representation of the binary tree, which is a list of node values in breadth-first order starting from the root (current node). If a node is at index i, its left child is always at 2i + 1, right child at 2i + 2, and parent at index floor((i - 1) / 2). None indicates absence of a node at that index. See example below for an illustration.

Return type [int | float | None]

Example:

```
>>> from binarytree import Node
>>>
>>> root = Node(1)
>>> root.left = Node(2)
>>> root.right = Node(3)
>>> root.left.right = Node(4)
>>>
>>> root.values
[1, 2, 3, None, 4]
```

3.2.2 Function: binarytree.build

binarytree.**build**(*values*)

Build a tree from list representation and return its root node.

Parameters **values** (*[int | float | None]*) – List representation of the binary tree, which is a list of node values in breadth-first order starting from the root (current node). If a node is at index i, its left child is always at 2i + 1, right child at 2i + 2, and parent at floor((i - 1) / 2). None indicates absence of a node at that index. See example below for an illustration.

Returns Root node of the binary tree.

Return type *binarytree.Node*

Raises *binarytree.exceptions.NodeNotFoundError* – If the list representation is malformed (e.g. a parent node is missing).

Example:

```
>>> from binarytree import build
>>>
>>> root = build([1, 2, 3, None, 4])
>>>
>>> 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*) – 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. If set to False, a perfect binary tree may still be generated by chance.

Returns Root node of the binary tree.

Return type *binarytree.Node*

Raises ***binarytree.exceptions.TreeHeightError*** – If height is invalid.

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
>>>
```

(continues on next page)

(continued from previous page)

```
>>> root = tree(height=20)
Traceback (most recent call last):
 ...
TreeHeightError: height must be an int 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*) – 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. If set to False, a perfect BST may still be generated by chance.

Returns Root node of the BST.

Return type *binarytree.Node*

Raises ***binarytree.exceptions.TreeHeightError*** – If height is invalid.

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):
 ...
TreeHeightError: height must be an int between 0 - 9
```

3.2.5 Function: binarytree.heap

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

Generate a random heap and return its root node.

Parameters

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

Returns Root node of the heap.

Return type *binarytree.Node*

Raises ***binarytree.exceptions.TreeHeightError*** – If height is invalid.

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):
 ...
TreeHeightError: height must be an int between 0 - 9
```

3.3 Exceptions

The page lists the exceptions raised by **binarytree**:

exception binarytree.exceptions.**BinaryTreeError**
Base (catch-all) binarytree exception.

exception binarytree.exceptions.**NodeIndexError**
Node index was invalid.

exception binarytree.exceptions.**NodeModifyError**
User tried to overwrite or delete the root node.

exception `binarytree.exceptions.`**`NodeNotFoundError`**
Node was missing from the binary tree.

exception `binarytree.exceptions.`**`NodeReferenceError`**
Node reference was invalid (e.g. cyclic reference).

exception `binarytree.exceptions.`**`NodeTypeError`**
Node was not an instance of *`binarytree.Node`*.

exception `binarytree.exceptions.`**`NodeValueError`**
Node value was not a number (e.g. int, float).

exception `binarytree.exceptions.`**`TreeHeightError`**
Tree height was invalid.

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 (good: "Add feature", bad: "Added feature").
- Correct and consistent style: Sphinx-compatible docstrings, correct snake and camel casing, and PEP8 compliance (see below).
- No classes/methods/functions with missing docstrings or commented-out lines. You can take a look at the source code on GitHub for examples.
- The test coverage remains at %100. 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 tests, 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).
- Compatibility with all supported 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. Before you begin feature work, git fetch or pull to ensure that your local branch has not diverged. If you see git conflicts and just want to start from scratch, run these commands:

```
~$ git checkout dev
~$ git fetch origin
~$ git reset --hard origin/dev  # THIS WILL WIPE ALL 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 must resolve all issues reported. If there is a good reason to ignore errors coming from a specific piece of code, visit here to see how to exclude the lines.

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 --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 --verbose --cov=binarytree --cov-report=html

# 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 an 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, thank you for your contribution!

Python Module Index

Index