
Python Tree Data

Release 2.7.3

c0fec0de

Dec 08, 2019

Contents

1 Installation	3
2 Introduction	5
2.1 Overview	5
2.2 Basics	6
2.3 Detach/Attach Protocol	7
2.4 Custom Separator	8
3 API	9
3.1 Node Classes	9
3.2 Tree Iteration	17
3.3 Tree Rendering	22
3.4 Searching	26
3.5 Cached Searching	30
3.6 Node Resolution	30
3.7 Tree Walking	32
3.8 Utilities	33
4 Importer	35
4.1 Dictionary Importer	35
4.2 JSON Importer	36
5 Exporter	37
5.1 Dictionary Exporter	37
5.2 JSON Exporter	38
5.3 Dot Exporter	39
6 Tricks	47
6.1 Read-only Tree	47
6.2 YAML Import/Export	49
6.3 Multidimensional Trees	51
7 Getting started	53
Python Module Index	57
Index	59



Simple, lightweight and extensible [Tree](#) data structure.

Feel free to [share](#) infos about your anytree project.

CHAPTER 1

Installation

To install the *anytree* module run:

```
pip install anytree
```

If you do not have write-permissions to the python installation, try:

```
pip install anytree --user
```


CHAPTER 2

Introduction

2.1 Overview

anytree is splitted into the following parts:

Node Classes

- `Node`: a simple tree node with at least a name attribute and any number of additional attributes.
- `AnyNode`: a generic tree node and any number of additional attributes.
- `NodeMixin`: extends any python class to a tree node.

Node Resolution

- `Resolver`: retrieve node via absolute or relative path.
- `Walker`: walk from one node to an other.

Tree Iteration Strategies

- `PreOrderIter`: iterate over tree using pre-order strategy
- `PostOrderIter`: iterate over tree using post-order strategy
- `LevelOrderIter`: iterate over tree using level-order strategy
- `LevelOrderGroupIter`: iterate over tree using level-order strategy returning group for every level
- `ZigZagGroupIter`: iterate over tree using level-order strategy returning group for every level

Tree Rendering

- **RenderTree using the following styles:**

- `Asciistyle`
- `ContStyle`
- `ContRoundStyle`
- `DoubleStyle`

2.2 Basics

The only tree relevant information is the *parent* attribute. If *None* the node is root node. If set to another node, the node becomes the child of it.

```
>>> from anytree import Node, RenderTree
>>> udo = Node("Udo")
>>> marc = Node("Marc")
>>> lian = Node("Lian", parent=marc)
>>> print(RenderTree(udo))
Node('/Udo')
>>> print(RenderTree(marc))
Node('/Marc')
└─ Node('/Marc/Lian')
```

Every node has a *children* attribute with a tuple of all children:

```
>>> udo.children
()
>>> marc.children
(Node('/Marc/Lian'),)
>>> lian.children
()
```

Single Node Attach

```
>>> marc.parent = udo
>>> print(RenderTree(udo))
Node('/Udo')
└─ Node('/Udo/Marc')
  └─ Node('/Udo/Marc/Lian')
```

Single Node Detach

To make a node to a root node, just set this attribute to *None*.

```
>>> marc.is_root
False
>>> marc.parent = None
>>> marc.is_root
True
```

Modify Multiple Child Nodes

```
>>> n = Node("n")
>>> a = Node("a", parent=n)
>>> b = Node("b", parent=n)
>>> c = Node("c", parent=n)
>>> d = Node("d")
>>> n.children
(Node('/n/a'), Node('/n/b'), Node('/n/c'))
```

Modifying the *children* attribute modifies multiple child nodes. It can be set to any iterable.

```
>>> n.children = [a, b]
>>> n.children
(Node('/n/a'), Node('/n/b'))
```

Node *c* is removed from the tree. In case of an existing reference, the node *c* does not vanish and is the root of its own tree.

```
>>> c
Node('/c')
```

Adding works likewise.

```
>>> d
Node('/d')
>>> n.children = [a, b, d]
>>> n.children
(Node('/n/a'), Node('/n/b'), Node('/n/d'))
>>> d
Node('/n/d')
```

2.3 Detach/Attach Protocol

A node class implementation might implement the notification slots `_pre_detach(parent)`, `_post_detach(parent)`, `_pre_attach(parent)`, `_post_attach(parent)`.

These methods are *protected* methods, intended to be overwritten by child classes of `NodeMixin/Node`. They are called on modifications of a nodes `parent` attribute. Never call them directly from API. This will corrupt the logic behind these methods.

```
>>> class NotifiedNode(Node):
...     def _pre_detach(self, parent):
...         print("_pre_detach", parent)
...     def _post_detach(self, parent):
...         print("_post_detach", parent)
...     def _pre_attach(self, parent):
...         print("_pre_attach", parent)
...     def _post_attach(self, parent):
...         print("_post_attach", parent)
```

Notification on attach:

```
>>> a = NotifiedNode("a")
>>> b = NotifiedNode("b")
>>> c = NotifiedNode("c")
>>> c.parent = a
_pre_attach NotifiedNode('/a')
_post_attach NotifiedNode('/a')
```

Notification on change:

```
>>> c.parent = b
_pre_detach NotifiedNode('/a')
_post_detach NotifiedNode('/a')
_pre_attach NotifiedNode('/b')
_post_attach NotifiedNode('/b')
```

If the parent equals the old value, the notification is not triggered:

```
>>> c.parent = b
```

Notification on detach:

```
>>> c.parent = None
._pre_detach NotifiedNode('/b')
._post_detach NotifiedNode('/b')
```

Important: An exception raised by `_pre_detach(parent)` and `_pre_attach(parent)` will **prevent** the tree structure to be updated. The node keeps the old state. An exception raised by `_post_detach(parent)` and `_post_attach(parent)` does **not rollback** the tree structure modification.

2.4 Custom Separator

By default a slash character (/) separates nodes. This separator can be overwritten:

```
>>> class MyNode(Node):
...     separator = " | "
>>> udo = MyNode("Udo")
>>> dan = MyNode("Dan", parent=udo)
>>> marc = MyNode("Marc", parent=udo)
>>> print(RenderTree(udo))
MyNode(' | Udo')
└── MyNode(' | Udo | Dan')
    └── MyNode(' | Udo | Marc')
```

The resolver takes the custom separator also into account:

```
>>> from anytree import Resolver
>>> r = Resolver()
>>> r.glob(udo, "|Udo|*")
[MyNode(' | Udo | Dan'), MyNode(' | Udo | Marc')]
```

CHAPTER 3

API

3.1 Node Classes

Node Classes.

- *AnyNode*: a generic tree node with any number of attributes.
- *Node*: a simple tree node with at least a name attribute and any number of additional attributes.
- *NodeMixin*: extends any python class to a tree node.

```
class anytree.node.anynode.AnyNode(parent=None, children=None, **kwargs)
Bases: anytree.node.nodemixin.NodeMixin, object
```

A generic tree node with any *kwargs*.

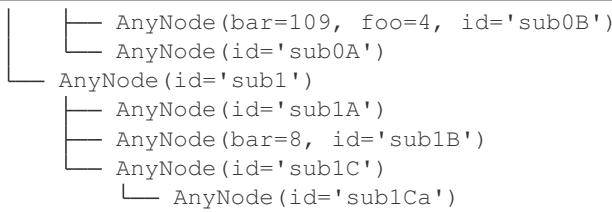
The *parent* attribute refers the parent node:

```
>>> from anytree import AnyNode, RenderTree
>>> root = AnyNode(id="root")
>>> s0 = AnyNode(id="sub0", parent=root)
>>> s0b = AnyNode(id="sub0B", parent=s0, foo=4, bar=109)
>>> s0a = AnyNode(id="sub0A", parent=s0)
>>> s1 = AnyNode(id="sub1", parent=root)
>>> s1a = AnyNode(id="sub1A", parent=s1)
>>> s1b = AnyNode(id="sub1B", parent=s1, bar=8)
>>> s1c = AnyNode(id="sub1C", parent=s1)
>>> s1ca = AnyNode(id="sub1Ca", parent=s1c)
```

```
>>> root
AnyNode(id='root')
>>> s0
AnyNode(id='sub0')
>>> print(RenderTree(root))
AnyNode(id='root')
|__ AnyNode(id='sub0')
```

(continues on next page)

(continued from previous page)



The same tree can be constructed by using the *children* attribute:

```

>>> root = AnyNode(id="root", children=[
...     AnyNode(id="sub0", children=[
...         AnyNode(id="sub0B", foo=4, bar=109),
...         AnyNode(id="sub0A"),
...     ]),
...     AnyNode(id="sub1", children=[
...         AnyNode(id="sub1A"),
...         AnyNode(id="sub1B", bar=8),
...         AnyNode(id="sub1C", children=[
...             AnyNode(id="sub1Ca"),
...         ]),
...     ]),
... ])
  
```

```

>>> print(RenderTree(root))
AnyNode(id='root')
└── AnyNode(id='sub0')
    ├── AnyNode(bar=109, foo=4, id='sub0B')
    └── AnyNode(id='sub0A')
└── AnyNode(id='sub1')
    ├── AnyNode(id='sub1A')
    ├── AnyNode(bar=8, id='sub1B')
    └── AnyNode(id='sub1C')
        └── AnyNode(id='sub1Ca')
  
```

Node attributes can be added, modified and deleted the pythonic way:

```

>>> root.new = 'a new attribute'
>>> s0b.bar = 110 # modified
>>> del s1b.bar
>>> print(RenderTree(root))
AnyNode(id='root', new='a new attribute')
└── AnyNode(id='sub0')
    ├── AnyNode(bar=109, foo=4, id='sub0B')
    └── AnyNode(id='sub0A')
└── AnyNode(id='sub1')
    ├── AnyNode(id='sub1A')
    ├── AnyNode(bar=8, id='sub1B')
    └── AnyNode(id='sub1C')
        └── AnyNode(id='sub1Ca')
  
```

```

class anytree.node.node.Node(name, parent=None, children=None, **kwargs)
Bases: anytree.node.nodemixin.NodeMixin, object
  
```

A simple tree node with a *name* and any *kwargs*.

The *parent* attribute refers the parent node:

```
>>> from anytree import Node, RenderTree
>>> root = Node("root")
>>> s0 = Node("sub0", parent=root)
>>> s0b = Node("sub0B", parent=s0, foo=4, bar=109)
>>> s0a = Node("sub0A", parent=s0)
>>> s1 = Node("sub1", parent=root)
>>> s1a = Node("sub1A", parent=s1)
>>> s1b = Node("sub1B", parent=s1, bar=8)
>>> s1c = Node("sub1C", parent=s1)
>>> s1ca = Node("sub1Ca", parent=s1c)
```

```
>>> print(RenderTree(root))
Node('/root')
└── Node('/root/sub0')
    ├── Node('/root/sub0/sub0B', bar=109, foo=4)
    └── Node('/root/sub0/sub0A')
└── Node('/root/sub1')
    ├── Node('/root/sub1/sub1A')
    ├── Node('/root/sub1/sub1B', bar=8)
    └── Node('/root/sub1/sub1C')
        └── Node('/root/sub1/sub1C/sub1Ca')
```

The same tree can be constructed by using the *children* attribute:

```
>>> root = Node("root", children=[
...     Node("sub0", children=[
...         Node("sub0B", bar=109, foo=4),
...         Node("sub0A", children=None),
...     ]),
...     Node("sub1", children=[
...         Node("sub1A"),
...         Node("sub1B", bar=8, children=[]),
...         Node("sub1C", children=[
...             Node("sub1Ca"),
...         ]),
...     ]),
... ])
```

```
>>> print(RenderTree(root))
Node('/root')
└── Node('/root/sub0')
    ├── Node('/root/sub0/sub0B', bar=109, foo=4)
    └── Node('/root/sub0/sub0A')
└── Node('/root/sub1')
    ├── Node('/root/sub1/sub1A')
    ├── Node('/root/sub1/sub1B', bar=8)
    └── Node('/root/sub1/sub1C')
        └── Node('/root/sub1/sub1C/sub1Ca')
```

class anytree.node.nodemixin.**NodeMixin**

Bases: `object`

separator = '/'

The `NodeMixin` class extends any Python class to a tree node.

The only tree relevant information is the *parent* attribute. If *None* the `NodeMixin` is root node. If set to another node, the `NodeMixin` becomes the child of it.

The `children` attribute can be used likewise. If `None` the `NodeMixin` has no children. The `children` attribute can be set to any iterable of `NodeMixin` instances. These instances become children of the node.

```
>>> from anytree import NodeMixin, RenderTree
>>> class MyBaseClass(object): # Just an example of a base class
...     foo = 4
>>> class MyClass(MyBaseClass, NodeMixin): # Add Node feature
...     def __init__(self, name, length, width, parent=None, children=None):
...         super(MyClass, self).__init__()
...         self.name = name
...         self.length = length
...         self.width = width
...         self.parent = parent
...         if children:
...             self.children = children
```

Construction via `parent`:

```
>>> my0 = MyClass('my0', 0, 0)
>>> my1 = MyClass('my1', 1, 0, parent=my0)
>>> my2 = MyClass('my2', 0, 2, parent=my0)
```

```
>>> for pre, _, node in RenderTree(my0):
...     treestr = u"%s%s" % (pre, node.name)
...     print(treestr.ljust(8), node.length, node.width)
my0      0 0
└── my1   1 0
    └── my2   0 2
```

Construction via `children`:

```
>>> my0 = MyClass('my0', 0, 0, children=[
...     MyClass('my1', 1, 0),
...     MyClass('my2', 0, 2),
... ])
```

```
>>> for pre, _, node in RenderTree(my0):
...     treestr = u"%s%s" % (pre, node.name)
...     print(treestr.ljust(8), node.length, node.width)
my0      0 0
└── my1   1 0
    └── my2   0 2
```

Both approaches can be mixed:

```
>>> my0 = MyClass('my0', 0, 0, children=[
...     MyClass('my1', 1, 0),
... ],
... )
>>> my2 = MyClass('my2', 0, 2, parent=my0)
```

```
>>> for pre, _, node in RenderTree(my0):
...     treestr = u"%s%s" % (pre, node.name)
...     print(treestr.ljust(8), node.length, node.width)
my0      0 0
└── my1   1 0
    └── my2   0 2
```

parent

Parent Node.

On set, the node is detached from any previous parent node and attached to the new node.

```
>>> from anytree import Node, RenderTree
>>> udo = Node("Udo")
>>> marc = Node("Marc")
>>> lian = Node("Lian", parent=marc)
>>> print(RenderTree(udo))
Node('/Udo')
>>> print(RenderTree(marc))
Node('/Marc')
└── Node('/Marc/Lian')
```

Attach

```
>>> marc.parent = udo
>>> print(RenderTree(udo))
Node('/Udo')
└── Node('/Udo/Marc')
    └── Node('/Udo/Marc/Lian')
```

Detach

To make a node to a root node, just set this attribute to *None*.

```
>>> marc.is_root
False
>>> marc.parent = None
>>> marc.is_root
True
```

children

All child nodes.

```
>>> from anytree import Node
>>> n = Node("n")
>>> a = Node("a", parent=n)
>>> b = Node("b", parent=n)
>>> c = Node("c", parent=n)
>>> n.children
(Node('/n/a'), Node('/n/b'), Node('/n/c'))
```

Modifying the children attribute modifies the tree.

Detach

The children attribute can be updated by setting to an iterable.

```
>>> n.children = [a, b]
>>> n.children
(Node('/n/a'), Node('/n/b'))
```

Node *c* is removed from the tree. In case of an existing reference, the node *c* does not vanish and is the root of its own tree.

```
>>> c
Node('/c')
```

Attach

```
>>> d = Node("d")
>>> d
Node('/d')
>>> n.children = [a, b, d]
>>> n.children
(Node('/n/a'), Node('/n/b'), Node('/n/d'))
>>> d
Node('/n/d')
```

Duplicate

A node can just be the children once. Duplicates cause a *TreeError*:

```
>>> n.children = [a, b, d, a]
Traceback (most recent call last):
...
anytree.node.exceptions.TreeError: Cannot add node Node('/n/a') multiple_
times as child.
```

path

Path of this *Node*.

```
>>> from anytree import Node
>>> udo = Node("Udo")
>>> marc = Node("Marc", parent=udo)
>>> lian = Node("Lian", parent=marc)
>>> udo.path
(Node('/Udo'),)
>>> marc.path
(Node('/Udo'), Node('/Udo/Marc'))
>>> lian.path
(Node('/Udo'), Node('/Udo/Marc'), Node('/Udo/Marc/Lian'))
```

iter_path_reverse()

Iterate up the tree from the current node.

```
>>> from anytree import Node
>>> udo = Node("Udo")
>>> marc = Node("Marc", parent=udo)
>>> lian = Node("Lian", parent=marc)
>>> for node in udo.iter_path_reverse():
...     print(node)
Node('/Udo')
>>> for node in marc.iter_path_reverse():
...     print(node)
Node('/Udo/Marc')
Node('/Udo')
>>> for node in lian.iter_path_reverse():
...     print(node)
Node('/Udo/Marc/Lian')
Node('/Udo/Marc')
Node('/Udo')
```

ancestors

All parent nodes and their parent nodes.

```
>>> from anytree import Node
>>> udo = Node("Udo")
>>> marc = Node("Marc", parent=udo)
>>> lian = Node("Lian", parent=marc)
>>> udo.ancestors
()
>>> marc.ancestors
(Node('/Udo'),)
>>> lian.ancestors
(Node('/Udo'), Node('/Udo/Marc'))
```

ancestors

All parent nodes and their parent nodes - see [ancestors](#).

The attribute *ancestors* is just a typo of *ancestors*. Please use *ancestors*. This attribute will be removed in the 3.0.0 release.

descendants

All child nodes and all their child nodes.

```
>>> from anytree import Node
>>> udo = Node("Udo")
>>> marc = Node("Marc", parent=udo)
>>> lian = Node("Lian", parent=marc)
>>> loui = Node("Loui", parent=marc)
>>> soe = Node("Soe", parent=lian)
>>> udo.descendants
(Node('/Udo/Marc'), Node('/Udo/Marc/Lian'), Node('/Udo/Marc/Lian/Soe'), Node(
    ↪ '/Udo/Marc/Loui'))
>>> marc.descendants
(Node('/Udo/Marc/Lian'), Node('/Udo/Marc/Lian/Soe'), Node('/Udo/Marc/Loui'))
>>> lian.descendants
(Node('/Udo/Marc/Lian/Soe'),)
```

root

Tree Root Node.

```
>>> from anytree import Node
>>> udo = Node("Udo")
>>> marc = Node("Marc", parent=udo)
>>> lian = Node("Lian", parent=marc)
>>> udo.root
Node('/Udo')
>>> marc.root
Node('/Udo')
>>> lian.root
Node('/Udo')
```

siblings

Tuple of nodes with the same parent.

```
>>> from anytree import Node
>>> udo = Node("Udo")
>>> marc = Node("Marc", parent=udo)
>>> lian = Node("Lian", parent=marc)
>>> loui = Node("Loui", parent=marc)
>>> lazy = Node("Lazy", parent=marc)
>>> udo.siblings
```

(continues on next page)

(continued from previous page)

```
()  
>>> marc.siblings  
()  
>>> lian.siblings  
(Node('/Udo/Marc/Loui'), Node('/Udo/Marc/Lazy'))  
>>> loui.siblings  
(Node('/Udo/Marc/Lian'), Node('/Udo/Marc/Lazy'))
```

leaves

Tuple of all leaf nodes.

```
>>> from anytree import Node  
>>> udo = Node("Udo")  
>>> marc = Node("Marc", parent=udo)  
>>> lian = Node("Lian", parent=marc)  
>>> loui = Node("Loui", parent=marc)  
>>> lazy = Node("Lazy", parent=marc)  
>>> udo.leaves  
(Node('/Udo/Marc/Lian'), Node('/Udo/Marc/Loui'), Node('/Udo/Marc/Lazy'))  
>>> marc.leaves  
(Node('/Udo/Marc/Lian'), Node('/Udo/Marc/Loui'), Node('/Udo/Marc/Lazy'))
```

is_leaf

Node has no children (External Node).

```
>>> from anytree import Node  
>>> udo = Node("Udo")  
>>> marc = Node("Marc", parent=udo)  
>>> lian = Node("Lian", parent=marc)  
>>> udo.is_leaf  
False  
>>> marc.is_leaf  
False  
>>> lian.is_leaf  
True
```

is_root

Node is tree root.

```
>>> from anytree import Node  
>>> udo = Node("Udo")  
>>> marc = Node("Marc", parent=udo)  
>>> lian = Node("Lian", parent=marc)  
>>> udo.is_root  
True  
>>> marc.is_root  
False  
>>> lian.is_root  
False
```

height

Number of edges on the longest path to a leaf *Node*.

```
>>> from anytree import Node  
>>> udo = Node("Udo")  
>>> marc = Node("Marc", parent=udo)
```

(continues on next page)

(continued from previous page)

```
>>> lian = Node("Lian", parent=marc)
>>> udo.height
2
>>> marc.height
1
>>> lian.height
0
```

depthNumber of edges to the root *Node*.

```
>>> from anytree import Node
>>> udo = Node("Udo")
>>> marc = Node("Marc", parent=udo)
>>> lian = Node("Lian", parent=marc)
>>> udo.depth
0
>>> marc.depth
1
>>> lian.depth
2
```

exception anytree.node.exceptions.**TreeError**

Bases: exceptions.RuntimeError

Tree Error.

exception anytree.node.exceptions.**LoopError**

Bases: anytree.node.exceptions.TreeError

Tree contains infinite loop.

3.2 Tree Iteration

Tree Iteration.

- *PreOrderIter*: iterate over tree using pre-order strategy (self, children)
- *PostOrderIter*: iterate over tree using post-order strategy (children, self)
- *LevelOrderIter*: iterate over tree using level-order strategy
- *LevelOrderGroupIter*: iterate over tree using level-order strategy returning group for every level
- *ZigZagGroupIter*: iterate over tree using level-order strategy returning group for every level

```
class anytree.iterators.preorderiter.PreOrderIter(node, filter_=None, stop=None,
                                                 maxlevel=None)
```

Bases: anytree.iterators.abstractiter.AbstractIter

Iterate over tree applying pre-order strategy starting at *node*.

Start at root and go-down until reaching a leaf node. Step upwards then, and search for the next leafs.

```
>>> from anytree import Node, RenderTree, AsciiStyle, PreOrderIter
>>> f = Node("f")
>>> b = Node("b", parent=f)
>>> a = Node("a", parent=b)
```

(continues on next page)

(continued from previous page)

```
>>> d = Node("d", parent=b)
>>> c = Node("c", parent=d)
>>> e = Node("e", parent=d)
>>> g = Node("g", parent=f)
>>> i = Node("i", parent=g)
>>> h = Node("h", parent=i)
>>> print(RenderTree(f, style=AsciiStyle()).by_attr())
f
| -- b
|   | -- a
|   +-- d
|     | -- c
|     +-- e
+-- g
  +-- i
    +-- h
>>> [node.name for node in PreOrderIter(f)]
['f', 'b', 'a', 'd', 'c', 'e', 'g', 'i', 'h']
>>> [node.name for node in PreOrderIter(f, maxlevel=3)]
['f', 'b', 'a', 'd', 'g', 'i']
>>> [node.name for node in PreOrderIter(f, filter_=lambda n: n.name not in ('e', 'g'))]
['f', 'b', 'a', 'd', 'c', 'i', 'h']
>>> [node.name for node in PreOrderIter(f, stop=lambda n: n.name == 'd')]
['f', 'b', 'a', 'g', 'i', 'h']
```

Base class for all iterators.

Iterate over tree starting at *node*.

Keyword Arguments

- **filter** – function called with every *node* as argument, *node* is returned if *True*.
- **stop** – stop iteration at *node* if *stop* function returns *True* for *node*.
- **maxlevel (int)** – maximum descending in the node hierarchy.

```
class anytree.iterators.postorderiter.PostOrderIter(node, filter_=None, stop=None,
                                                    maxlevel=None)
```

Bases: anytree.iterators.abstractiter.AbstractIter

Iterate over tree applying post-order strategy starting at *node*.

```
>>> from anytree import Node, RenderTree, AsciiStyle, PostOrderIter
>>> f = Node("f")
>>> b = Node("b", parent=f)
>>> a = Node("a", parent=b)
>>> d = Node("d", parent=b)
>>> c = Node("c", parent=d)
>>> e = Node("e", parent=d)
>>> g = Node("g", parent=f)
>>> i = Node("i", parent=g)
>>> h = Node("h", parent=i)
>>> print(RenderTree(f, style=AsciiStyle()).by_attr())
f
| -- b
|   | -- a
|   +-- d
```

(continues on next page)

(continued from previous page)

```

|       | -- c
|       +-- e
+-- g
    +-- i
        +-- h
>>> [node.name for node in PostOrderIter(f)]
['a', 'c', 'e', 'd', 'b', 'h', 'i', 'g', 'f']
>>> [node.name for node in PostOrderIter(f, maxlevel=3)]
['a', 'd', 'b', 'i', 'g', 'f']
>>> [node.name for node in PostOrderIter(f, filter_=lambda n: n.name not in ('e', 'g'))]
['a', 'c', 'd', 'b', 'h', 'i', 'f']
>>> [node.name for node in PostOrderIter(f, stop=lambda n: n.name == 'd')]
['a', 'b', 'h', 'i', 'g', 'f']

```

Base class for all iterators.

Iterate over tree starting at *node*.

Keyword Arguments

- **filter** – function called with every *node* as argument, *node* is returned if *True*.
- **stop** – stop iteration at *node* if *stop* function returns *True* for *node*.
- **maxlevel (int)** – maximum descending in the node hierarchy.

```
class anytree.iterators.levelorderiter.LevelOrderIter(node, filter_=None, stop=None, maxlevel=None)
```

Bases: anytree.iterators.abstractiter.AbstractIter

Iterate over tree applying level-order strategy starting at *node*.

```

>>> from anytree import Node, RenderTree, AsciiStyle, LevelOrderIter
>>> f = Node("f")
>>> b = Node("b", parent=f)
>>> a = Node("a", parent=b)
>>> d = Node("d", parent=b)
>>> c = Node("c", parent=d)
>>> e = Node("e", parent=d)
>>> g = Node("g", parent=f)
>>> i = Node("i", parent=g)
>>> h = Node("h", parent=i)
>>> print(RenderTree(f, style=AsciiStyle()).by_attr())
f
|-- b
|   |-- a
|   +-- d
|       |-- c
|       +-- e
+-- g
    +-- i
        +-- h
>>> [node.name for node in LevelOrderIter(f)]
['f', 'b', 'g', 'a', 'd', 'i', 'c', 'e', 'h']
>>> [node.name for node in LevelOrderIter(f, maxlevel=3)]
['f', 'b', 'g', 'a', 'd', 'i']
>>> [node.name for node in LevelOrderIter(f, filter_=lambda n: n.name not in ('e', 'g'))]

```

(continues on next page)

(continued from previous page)

```
[ 'f', 'b', 'a', 'd', 'i', 'c', 'h']
>>> [node.name for node in LevelOrderIter(f, stop=lambda n: n.name == 'd') ]
[ 'f', 'b', 'g', 'a', 'i', 'h']
```

Base class for all iterators.

Iterate over tree starting at *node*.

Keyword Arguments

- **filter** – function called with every *node* as argument, *node* is returned if *True*.
- **stop** – stop iteration at *node* if *stop* function returns *True* for *node*.
- **maxlevel (int)** – maximum descending in the node hierarchy.

```
class anytree.iterators.levelordergroupiter.LevelOrderGroupIter(node,      fil-
                                                               ter=None,
                                                               stop=None,
                                                               maxlevel=None)
```

Bases: anytree.iterators.abstractiter.AbstractIter

Iterate over tree applying level-order strategy with grouping starting at *node*.

Return a tuple of nodes for each level. The first tuple contains the nodes at level 0 (always *node*). The second tuple contains the nodes at level 1 (children of *node*). The next level contains the children of the children, and so on.

```
>>> from anytree import Node, RenderTree, AsciiStyle, LevelOrderGroupIter
>>> f = Node("f")
>>> b = Node("b", parent=f)
>>> a = Node("a", parent=b)
>>> d = Node("d", parent=b)
>>> c = Node("c", parent=d)
>>> e = Node("e", parent=d)
>>> g = Node("g", parent=f)
>>> i = Node("i", parent=g)
>>> h = Node("h", parent=i)
>>> print(RenderTree(f, style=AsciiStyle()).by_attr())
f
|-- b
|   |-- a
|   +-- d
|       |-- c
|       +-- e
+-- g
    +-- i
        +-- h
>>> [[node.name for node in children] for children in LevelOrderGroupIter(f)]
[['f'], ['b', 'g'], ['a', 'd', 'i'], ['c', 'e', 'h']]
>>> [[node.name for node in children] for children in LevelOrderGroupIter(f, ↴maxlevel=3)]
[['f'], ['b', 'g'], ['a', 'd', 'i']]
>>> [[node.name for node in children]
...   for children in LevelOrderGroupIter(f, filter_=lambda n: n.name not in ('e', ↴'g'))]
[['f'], ['b'], ['a', 'd', 'i'], ['c', 'h']]
>>> [[node.name for node in children]
...   for children in LevelOrderGroupIter(f, stop=lambda n: n.name == 'd')]]
[['f'], ['b', 'g'], ['a', 'i'], ['h']]
```

Base class for all iterators.

Iterate over tree starting at *node*.

Keyword Arguments

- **filter** – function called with every *node* as argument, *node* is returned if *True*.
- **stop** – stop iteration at *node* if *stop* function returns *True* for *node*.
- **maxlevel** (*int*) – maximum descending in the node hierarchy.

```
class anytree.iterators.zigzaggroupiter.ZigZagGroupIter(node,      filter_=None,
                                                       stop=None,
                                                       maxlevel=None)
```

Bases: anytree.iterators.abstractiter.AbstractIter

Iterate over tree applying Zig-Zag strategy with grouping starting at *node*.

Return a tuple of nodes for each level. The first tuple contains the nodes at level 0 (always *node*). The second tuple contains the nodes at level 1 (children of *node*) in reversed order. The next level contains the children of the children in forward order, and so on.

```
>>> from anytree import Node, RenderTree, AsciiStyle
>>> f = Node("f")
>>> b = Node("b", parent=f)
>>> a = Node("a", parent=b)
>>> d = Node("d", parent=b)
>>> c = Node("c", parent=d)
>>> e = Node("e", parent=d)
>>> g = Node("g", parent=f)
>>> i = Node("i", parent=g)
>>> h = Node("h", parent=i)
>>> print(RenderTree(f, style=AsciiStyle()).by_attr())
f
|--- b
|   |--- a
|   +--- d
|       |--- c
|       +--- e
+--- g
    +--- i
        +--- h
>>> [[node.name for node in children] for children in ZigZagGroupIter(f)]
[['f'], ['g', 'b'], ['a', 'd', 'i'], ['h', 'e', 'c']]
>>> [[node.name for node in children] for children in ZigZagGroupIter(f, maxLevel=3)]
[['f'], ['g', 'b'], ['a', 'd', 'i']]
>>> [[node.name for node in children]
...   for children in ZigZagGroupIter(f, filter_=lambda n: n.name not in ('e', 'g'))]
[['f'], ['b'], ['a', 'd', 'i'], ['h', 'c']]
>>> [[node.name for node in children]
...   for children in ZigZagGroupIter(f, stop=lambda n: n.name == 'd')]
[['f'], ['g', 'b'], ['a', 'i'], ['h']]
```

Base class for all iterators.

Iterate over tree starting at *node*.

Keyword Arguments

- **filter** – function called with every *node* as argument, *node* is returned if *True*.

- **stop** – stop iteration at *node* if *stop* function returns *True* for *node*.
- **maxlevel** (*int*) – maximum descending in the node hierarchy.

3.3 Tree Rendering

Tree Rendering.

- **RenderTree** using the following styles:

- *AsciiStyle*
- *ContStyle*
- *ContRoundStyle*
- *DoubleStyle*

class anytree.render.Row(*pre*, *fill*, *node*)

Bases: *tuple*

Create new instance of Row(*pre*, *fill*, *node*)

fill

Alias for field number 1

node

Alias for field number 2

pre

Alias for field number 0

class anytree.render.AbstractStyle(*vertical*, *cont*, *end*)

Bases: *object*

Tree Render Style.

Parameters

- **vertical** – Sign for vertical line.
- **cont** – Chars for a continued branch.
- **end** – Chars for the last branch.

empty

Empty string as placeholder.

class anytree.render.AsciiStyle

Bases: anytree.render.AbstractStyle

Ascii style.

```
>>> from anytree import Node, RenderTree
>>> root = Node("root")
>>> s0 = Node("sub0", parent=root)
>>> s0b = Node("sub0B", parent=s0)
>>> s0a = Node("sub0A", parent=s0)
>>> s1 = Node("sub1", parent=root)
```

```
>>> print(RenderTree(root, style=AsciiStyle()))
Node('/root')
|-- Node('/root/sub0')
|   |-- Node('/root/sub0/sub0B')
|   +-- Node('/root/sub0/sub0A')
+-- Node('/root/sub1')
```

class anytree.render.**ContStyle**
 Bases: anytree.render.AbstractStyle

Continued style, without gaps.

```
>>> from anytree import Node, RenderTree
>>> root = Node("root")
>>> s0 = Node("sub0", parent=root)
>>> s0b = Node("sub0B", parent=s0)
>>> s0a = Node("sub0A", parent=s0)
>>> s1 = Node("sub1", parent=root)
```

```
>>> print(RenderTree(root, style=ContStyle()))
Node('/root')
└── Node('/root/sub0')
    ├── Node('/root/sub0/sub0B')
    └── Node('/root/sub0/sub0A')
└── Node('/root/sub1')
```

class anytree.render.**ContRoundStyle**
 Bases: anytree.render.AbstractStyle

Continued style, without gaps, round edges.

```
>>> from anytree import Node, RenderTree
>>> root = Node("root")
>>> s0 = Node("sub0", parent=root)
>>> s0b = Node("sub0B", parent=s0)
>>> s0a = Node("sub0A", parent=s0)
>>> s1 = Node("sub1", parent=root)
```

```
>>> print(RenderTree(root, style=ContRoundStyle()))
Node('/root')
└── Node('/root/sub0')
    ├── Node('/root/sub0/sub0B')
    └── Node('/root/sub0/sub0A')
└── Node('/root/sub1')
```

class anytree.render.**DoubleStyle**
 Bases: anytree.render.AbstractStyle

Double line style, without gaps.

```
>>> from anytree import Node, RenderTree
>>> root = Node("root")
>>> s0 = Node("sub0", parent=root)
>>> s0b = Node("sub0B", parent=s0)
>>> s0a = Node("sub0A", parent=s0)
>>> s1 = Node("sub1", parent=root)
```

```
>>> print(RenderTree(root, style=DoubleStyle))
Node('/root')
Node('/root/sub0')
Node('/root/sub0/sub0B')
Node('/root/sub0/sub0A')
Node('/root/sub1')
```

class anytree.render.RenderTree(*node*, *style*=*ContStyle()*, *childiter*=*<type 'list'>*)
Bases: *object*

Render tree starting at *node*.

Keyword Arguments

- **style** (*AbstractStyle*) – Render Style.
- **childiter** – Child iterator.

RenderTree is an iterator, returning a tuple with 3 items:

pre tree prefix.

fill filling for multiline entries.

node *NodeMixin* object.

It is up to the user to assemble these parts to a whole.

```
>>> from anytree import Node, RenderTree
>>> root = Node("root", lines=["c0fe", "c0de"])
>>> s0 = Node("sub0", parent=root, lines=["ha", "ba"])
>>> s0b = Node("sub0B", parent=s0, lines=["1", "2", "3"])
>>> s0a = Node("sub0A", parent=s0, lines=["a", "b"])
>>> s1 = Node("sub1", parent=root, lines=["Z"])
```

Simple one line:

```
>>> for pre, __, node in RenderTree(root):
...     print("%s%s" % (pre, node.name))
root
└── sub0
    ├── sub0B
    │   ├── 1
    │   ├── 2
    │   └── 3
    └── sub0A
        ├── a
        └── b
└── sub1
```

Multiline:

```
>>> for pre, fill, node in RenderTree(root):
...     print("%s%s" % (pre, node.lines[0]))
...     for line in node.lines[1:]:
...         print("%s%s" % (fill, line))
c0fe
c0de
└── ha
    └── ba
        ├── 1
        ├── 2
        └── 3
    └── sub0A
        ├── a
        └── b
└── sub1
```

The `childiter` is responsible for iterating over child nodes at the same level. An reversed order can be achived by using `reversed`.

```
>>> for row in RenderTree(root, childiter=reversed):
...     print("%s%s" % (row.pre, row.node.name))
root
└── sub1
    └── sub0
        ├── sub0A
        └── sub0B
```

Or writing your own sort function:

```
>>> def mysort(items):
...     return sorted(items, key=lambda item: item.name)
>>> for row in RenderTree(root, childiter=mysort):
...     print("%s%s" % (row.pre, row.node.name))
root
└── sub0
    ├── sub0A
    └── sub0B
    └── sub1
```

`by_attr` simplifies attribute rendering and supports multiline:

```
>>> print(RenderTree(root).by_attr())
root
└── sub0
    ├── sub0B
    └── sub0A
    └── sub1
>>> print(RenderTree(root).by_attr("lines"))
c0fe
c0de
    └── ha
        └── ba
            └── 1
                └── 2
                └── 3
            └── a
            └── b
    └── Z
```

And can be a function:

```
>>> print(RenderTree(root).by_attr(lambda n: " ".join(n.lines)))
c0fe c0de
    └── ha
        └── ba
            └── 1 2 3
            └── a b
    └── Z
```

`by_attr(attrname='name')`

Return rendered tree with node attribute `attrname`.

```
>>> from anytree import AnyNode, RenderTree
>>> root = AnyNode(id="root")
>>> s0 = AnyNode(id="sub0", parent=root)
```

(continues on next page)

(continued from previous page)

```
>>> s0b = AnyNode(id="sub0B", parent=s0, foo=4, bar=109)
>>> s0a = AnyNode(id="sub0A", parent=s0)
>>> s1 = AnyNode(id="sub1", parent=root)
>>> s1a = AnyNode(id="sub1A", parent=s1)
>>> s1b = AnyNode(id="sub1B", parent=s1, bar=8)
>>> s1c = AnyNode(id="sub1C", parent=s1)
>>> s1ca = AnyNode(id="sub1Ca", parent=s1c)
>>> print(RenderTree(root).by_attr('id'))
root
└── sub0
    ├── sub0B
    └── sub0A
└── sub1
    ├── sub1A
    ├── sub1B
    └── sub1C
        └── sub1Ca
```

3.4 Searching

Node Searching.

Note: You can speed-up node searching, by installing <https://pypi.org/project/fastcache/> and using `cachedsearch`.

`anytree.search.findall(node, filter_=None, stop=None, maxlevel=None, mincount=None, maxcount=None)`

Search nodes matching `filter_` but stop at `maxlevel` or `stop`.

Return tuple with matching nodes.

Parameters `node` – top node, start searching.

Keyword Arguments

- `filter` – function called with every `node` as argument, `node` is returned if `True`.
- `stop` – stop iteration at `node` if `stop` function returns `True` for `node`.
- `maxlevel` (`int`) – maximum descending in the node hierarchy.
- `mincount` (`int`) – minimum number of nodes.
- `maxcount` (`int`) – maximum number of nodes.

Example tree:

```
>>> from anytree import Node, RenderTree, AsciiStyle
>>> f = Node("f")
>>> b = Node("b", parent=f)
>>> a = Node("a", parent=b)
>>> d = Node("d", parent=b)
>>> c = Node("c", parent=d)
>>> e = Node("e", parent=d)
>>> g = Node("g", parent=f)
>>> i = Node("i", parent=g)
>>> h = Node("h", parent=i)
```

(continues on next page)

(continued from previous page)

```
>>> print(RenderTree(f, style=AsciiStyle()).by_attr())
f
|--- b
|   |--- a
|   +--- d
|       |--- c
|       +--- e
+--- g
    +--- i
        +--- h
```

```
>>>.findall(f, filter_=lambda node: node.name in ("a", "b"))
(Node('/f/b'), Node('/f/b/a'))
>>>.findall(f, filter_=lambda node: d in node.path)
(Node('/f/b/d'), Node('/f/b/d/c'), Node('/f/b/d/e'))
```

The number of matches can be limited:

```
>>>.findall(f, filter_=lambda node: d in node.path, mincount=4)  # doctest:_
˓→+ELLIPSIS
Traceback (most recent call last):
...
anytree.search.CountError: Expecting at least 4 elements, but found 3. ... Node('/
˓→f/b/d/e'))
>>>.findall(f, filter_=lambda node: d in node.path, maxcount=2)  # doctest:_
˓→+ELLIPSIS
Traceback (most recent call last):
...
anytree.search.CountError: Expecting 2 elements at maximum, but found 3. ... Node(
˓→'/f/b/d/e'))
```

`anytree.search.findall_by_attr(node, value, name='name', maxlevel=None, mincount=None, maxcount=None)`

Search nodes with attribute `name` having `value` but stop at `maxlevel`.

Return tuple with matching nodes.

Parameters

- `node` – top node, start searching.
- `value` – value which need to match

Keyword Arguments

- `name` (`str`) – attribute name need to match
- `maxlevel` (`int`) – maximum descending in the node hierarchy.
- `mincount` (`int`) – minimum number of nodes.
- `maxcount` (`int`) – maximum number of nodes.

Example tree:

```
>>> from anytree import Node, RenderTree, AsciiStyle
>>> f = Node("f")
>>> b = Node("b", parent=f)
>>> a = Node("a", parent=b)
>>> d = Node("d", parent=b)
```

(continues on next page)

(continued from previous page)

```
>>> c = Node("c", parent=d)
>>> e = Node("e", parent=d)
>>> g = Node("g", parent=f)
>>> i = Node("i", parent=g)
>>> h = Node("h", parent=i)
>>> print(RenderTree(f, style=AsciiStyle()).by_attr())
f
| -- b
|   | -- a
|   +-- d
|     | -- c
|     +-- e
+-- g
  +-- i
    +-- h
```

```
>>>.findall_by_attr(f, "d")
(Node('/f/b/d'),)
```

`anytree.search.find(node, filter_=None, stop=None, maxlevel=None)`

Search for *single* node matching `filter_` but stop at `maxlevel` or `stop`.

Return matching node.

Parameters `node` – top node, start searching.

Keyword Arguments

- `filter` – function called with every `node` as argument, `node` is returned if `True`.
- `stop` – stop iteration at `node` if `stop` function returns `True` for `node`.
- `maxlevel (int)` – maximum descending in the node hierarchy.

Example tree:

```
>>> from anytree import Node, RenderTree, AsciiStyle
>>> f = Node("f")
>>> b = Node("b", parent=f)
>>> a = Node("a", parent=b)
>>> d = Node("d", parent=b)
>>> c = Node("c", parent=d)
>>> e = Node("e", parent=d)
>>> g = Node("g", parent=f)
>>> i = Node("i", parent=g)
>>> h = Node("h", parent=i)
>>> print(RenderTree(f, style=AsciiStyle()).by_attr())
f
| -- b
|   | -- a
|   +-- d
|     | -- c
|     +-- e
+-- g
  +-- i
    +-- h
```

```
>>> find(f, lambda node: node.name == "d")
Node('/f/b/d')
>>> find(f, lambda node: node.name == "z")
>>> find(f, lambda node: b in node.path)  # doctest: +ELLIPSIS
Traceback (most recent call last):
...
anytree.search.CountError: Expecting 1 elements at maximum, but found 5. (Node('/
↳ f/b')... Node('/f/b/d/e'))
```

`anytree.search.find_by_attr(node, value, name='name', maxlevel=None)`

Search for *single* node with attribute *name* having *value* but stop at *maxlevel*.

Return tuple with matching nodes.

Parameters

- **node** – top node, start searching.
- **value** – value which need to match

Keyword Arguments

- **name** (*str*) – attribute name need to match
- **maxlevel** (*int*) – maximum descending in the node hierarchy.

Example tree:

```
>>> from anytree import Node, RenderTree, AsciiStyle
>>> f = Node("f")
>>> b = Node("b", parent=f)
>>> a = Node("a", parent=b)
>>> d = Node("d", parent=b)
>>> c = Node("c", parent=d, foo=4)
>>> e = Node("e", parent=d)
>>> g = Node("g", parent=f)
>>> i = Node("i", parent=g)
>>> h = Node("h", parent=i)
>>> print(RenderTree(f, style=AsciiStyle()).by_attr())
f
| -- b
|   | -- a
|   +-- d
|     | -- c
|     +-- e
+-- g
  +-- i
    +-- h
```

```
>>> find_by_attr(f, "d")
Node('/f/b/d')
>>> find_by_attr(f, name="foo", value=4)
Node('/f/b/d/c', foo=4)
>>> find_by_attr(f, name="foo", value=8)
```

exception `anytree.search.CountError(msg, result)`

Bases: `exceptions.RuntimeError`

Error raised on *mincount* or *maxcount* mismatch.

3.5 Cached Searching

Node Searching with Cache.

Note: These functions require <https://pypi.org/project/fastcache/>, otherwise caching is not active.

`anytree.cachedsearch.findall(*args, **kwargs)`

Identical to `search.findall` but cached.

`anytree.cachedsearch.findall_by_attr(*args, **kwargs)`

Identical to `search.findall_by_attr` but cached.

`anytree.cachedsearch.find(*args, **kwargs)`

Identical to `search.find` but cached.

`anytree.cachedsearch.find_by_attr(*args, **kwargs)`

Identical to `search.find_by_attr` but cached.

3.6 Node Resolution

`class anytree.resolver.Resolver(pathattr='name')`

Bases: `object`

Resolve `NodeMixin` paths using attribute `pathattr`.

`get(node, path)`

Return instance at `path`.

An example module tree:

```
>>> from anytree import Node
>>> top = Node("top", parent=None)
>>> sub0 = Node("sub0", parent=top)
>>> sub0sub0 = Node("sub0sub0", parent=sub0)
>>> sub0sub1 = Node("sub0sub1", parent=sub0)
>>> sub1 = Node("sub1", parent=top)
```

A resolver using the `name` attribute:

```
>>> r = Resolver('name')
```

Relative paths:

```
>>> r.get(top, "sub0/sub0sub0")
Node('/top/sub0/sub0sub0')
>>> r.get(sub1, ".")
Node('/top')
>>> r.get(sub1, ".../sub0/sub0sub1")
Node('/top/sub0/sub0sub1')
>>> r.get(sub1, ".")
Node('/top/sub1')
>>> r.get(sub1, "")
Node('/top/sub1')
>>> r.get(top, "sub2")
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```

...
anytree.resolver.ChildResolverError: Node('/top') has no child sub2. Children ↵
→are: 'sub0', 'sub1'.

```

Absolute paths:

```

>>> r.get(sub0sub0, "/top")
Node('/top')
>>> r.get(sub0sub0, "/top/sub0")
Node('/top/sub0')
>>> r.get(sub0sub0, "/")
Traceback (most recent call last):
...
anytree.resolver.ResolverError: root node missing. root is '/top'.
>>> r.get(sub0sub0, "/bar")
Traceback (most recent call last):
...
anytree.resolver.ResolverError: unknown root node '/bar'. root is '/top'.

```

glob (*node, path*)

Return instances at *path* supporting wildcards.

Behaves identical to [get](#), but accepts wildcards and returns a list of found nodes.

- * matches any characters, except '/'.
- ? matches a single character, except '/'.

An example module tree:

```

>>> from anytree import Node
>>> top = Node("top", parent=None)
>>> sub0 = Node("sub0", parent=top)
>>> sub0sub0 = Node("sub0", parent=sub0)
>>> sub0sub1 = Node("sub1", parent=sub0)
>>> sub1 = Node("sub1", parent=top)
>>> sub1sub0 = Node("sub0", parent=sub1)

```

A resolver using the *name* attribute:

```
>>> r = Resolver('name')
```

Relative paths:

```

>>> r.glob(top, "sub0/sub?")
[Node('/top/sub0/sub0'), Node('/top/sub0/sub1')]
>>> r.glob(sub1, ".../*")
[Node('/top/sub0'), Node('/top/sub1')]
>>> r.glob(top, "*/*")
[Node('/top/sub0/sub0'), Node('/top/sub0/sub1'), Node('/top/sub1/sub0')]
>>> r.glob(top, "*/*sub0")
[Node('/top/sub0/sub0'), Node('/top/sub1/sub0')]
>>> r.glob(top, "sub1/*")
Traceback (most recent call last):
...
anytree.resolver.ChildResolverError: Node('/top/sub1') has no child sub1. ↵
→Children are: 'sub0'.

```

Non-matching wildcards are no error:

```
>>> r.glob(top, "bar*")
[]
>>> r.glob(top, "sub2")
Traceback (most recent call last):
...
anytree.resolver.ChildResolverError: Node('/top') has no child sub2. Children
←are: 'sub0', 'sub1'.
```

Absolute paths:

```
>>> r.glob(sub0sub0, "/top/*")
[Node('/top/sub0'), Node('/top/sub1')]
>>> r.glob(sub0sub0, "/")
Traceback (most recent call last):
...
anytree.resolver.ResolverError: root node missing. root is '/top'.
>>> r.glob(sub0sub0, "/bar")
Traceback (most recent call last):
...
anytree.resolver.ResolverError: unknown root node '/bar'. root is '/top'.
```

static is_wildcard(path)

Return *True* if *path* is a wildcard.

exception anytree.resolver.ResolverError(node, child, msg)

Bases: exceptions.RuntimeError

Resolve Error at *node* handling *child*.

exception anytree.resolver.ChildResolverError(node, child, pathattr)

Bases: *anytree.resolver.ResolverError*

Child Resolve Error at *node* handling *child*.

3.7 Tree Walking

class anytree.walker.Walker

Bases: *object*

Walk from one node to another.

walk(start, end)

Walk from *start* node to *end* node.

Returns *upwards* is a list of nodes to go upward to. *common* top node. *downwards* is a list of nodes to go downward to.

Return type (upwards, common, downwards)

Raises *WalkError* – on no common root node.

```
>>> from anytree import Node, RenderTree, AsciiStyle
>>> f = Node("f")
>>> b = Node("b", parent=f)
>>> a = Node("a", parent=b)
>>> d = Node("d", parent=b)
>>> c = Node("c", parent=d)
>>> e = Node("e", parent=d)
```

(continues on next page)

(continued from previous page)

```
>>> g = Node("g", parent=f)
>>> i = Node("i", parent=g)
>>> h = Node("h", parent=i)
>>> print(RenderTree(f, style=AsciiStyle()))
Node('/f')
| -- Node('/f/b')
|   | -- Node('/f/b/a')
|   +-- Node('/f/b/d')
|     | -- Node('/f/b/d/c')
|     +-- Node('/f/b/d/e')
+-- Node('/f/g')
  +-- Node('/f/g/i')
    +-- Node('/f/g/i/h')
```

Create a walker:

```
>>> w = Walker()
```

This class is made for walking:

```
>>> w.walk(f, f)
((), Node('/f'), ())
>>> w.walk(f, b)
((), Node('/f'), (Node('/f/b'),))
>>> w.walk(b, f)
((Node('/f/b'),), Node('/f'), ())
>>> w.walk(h, e)
((Node('/f/g/i/h'), Node('/f/g/i'), Node('/f/g')), Node('/f'), (Node('/f/b'), Node('/f/b/d'), Node('/f/b/d/e')))
>>> w.walk(d, e)
((), Node('/f/b/d'), (Node('/f/b/d/e'),))
```

For a proper walking the nodes need to be part of the same tree:

```
>>> w.walk(Node("a"), Node("b"))
Traceback (most recent call last):
...
anytree.walker.WalkError: Node('/a') and Node('/b') are not part of the same tree.
```

exception anytree.walker.WalkError

Bases: exceptions.RuntimeError

Walk Error.

3.8 Utilities

Utilities.

anytree.util.commonancestors(*nodes)

Determine common ancestors of *nodes*.

```
>>> from anytree import Node
>>> udo = Node("Udo")
>>> marc = Node("Marc", parent=udo)
```

(continues on next page)

(continued from previous page)

```
>>> lian = Node("Lian", parent=marc)
>>> dan = Node("Dan", parent=udo)
>>> jet = Node("Jet", parent=dan)
>>> jan = Node("Jan", parent=dan)
>>> joe = Node("Joe", parent=dan)
```

```
>>> commonancestors(jet, joe)
(Node('/Udo'), Node('/Udo/Dan'))
>>> commonancestors(jet, marc)
(Node('/Udo'),)
>>> commonancestors(jet)
(Node('/Udo'), Node('/Udo/Dan'))
>>> commonancestors()
()
```

anytree.util.leftsibling(*node*)

Return Left Sibling of *node*.

```
>>> from anytree import Node
>>> dan = Node("Dan")
>>> jet = Node("Jet", parent=dan)
>>> jan = Node("Jan", parent=dan)
>>> joe = Node("Joe", parent=dan)
>>> leftsibling(dan)
>>> leftsibling(jet)
>>> leftsibling(jan)
Node('/Dan/Jet')
>>> leftsibling(joe)
Node('/Dan/Jan')
```

anytree.util.rightsibling(*node*)

Return Right Sibling of *node*.

```
>>> from anytree import Node
>>> dan = Node("Dan")
>>> jet = Node("Jet", parent=dan)
>>> jan = Node("Jan", parent=dan)
>>> joe = Node("Joe", parent=dan)
>>> rightsibling(dan)
>>> rightsibling(jet)
Node('/Dan/Jan')
>>> rightsibling(jan)
Node('/Dan/Jan')
>>> rightsibling(joe)
```

CHAPTER 4

Importer

One fundamental idea behind *anytree* is the common tree node data structure, which can be imported from different formats and exported to different formats.

Available importers:

4.1 Dictionary Importer

```
class anytree.importer.dictimporter.DictImporter(nodecls=<class  
'anytree.node.anynode.AnyNode'>)
```

Bases: `object`

Import Tree from dictionary.

Every dictionary is converted to an instance of `nodecls`. The dictionaries listed in the `children` attribute are converted likewise and added as children.

Keyword Arguments `nodecls` – class used for nodes.

```
>>> from anytree.importer import DictImporter  
>>> from anytree import RenderTree  
>>> importer = DictImporter()  
>>> data = {  
...     'a': 'root',  
...     'children': [{  
...         'a': 'sub0',  
...         'children': [{  
...             'a': 'sub0A',  
...             'b': 'foo'},  
...             {'a': 'sub0B'}]  
...         'a': 'sub1'}]}  
>>> root = importer.import_(data)  
>>> print(RenderTree(root))
```

```
AnyNode(a='root')  
|   AnyNode(a='sub0')  
|       AnyNode(a='sub0A', b='foo')  
|       AnyNode(a='sub0B')  
|   AnyNode(a='sub1')
```

```
import_(data)
Import tree from data.
```

4.2 JSON Importer

```
class anytree.importer.jsonimporter.JsonImporter(dictimporter=None, **kwargs)
Bases: object
```

Import Tree from JSON.

The JSON is read and converted to a dictionary via *dictimporter*.

Keyword Arguments

- **dictimporter** – Dictionary Importer used (see *DictImporter*).
- **kwargs** – All other arguments are passed to `json.load/json.loads`. See documentation for reference.

```
>>> from anytree.importer import JsonImporter
>>> from anytree import RenderTree
>>> importer = JsonImporter()
>>> data = '''
... {
...     "a": "root",
...     "children": [
...         {
...             "a": "sub0",
...             "children": [
...                 {
...                     "a": "sub0A",
...                     "b": "foo"
...                 },
...                 {
...                     "a": "sub0B"
...                 }
...             ]
...         },
...         {
...             "a": "sub1"
...         }
...     ]
... }'''
>>> root = importer.import_(data)
>>> print(RenderTree(root))
AnyNode(a='root')
└── AnyNode(a='sub0')
    ├── AnyNode(a='sub0A', b='foo')
    └── AnyNode(a='sub0B')
└── AnyNode(a='sub1')
```

```
import_(data)
Read JSON from data.
read(filehandle)
Read JSON from filehandle.
```

Importer missing? File a request here: [Issues](#).

CHAPTER 5

Exporter

One fundamental idea behind *anytree* is the common tree node data structure, which can be imported from different formats and exported to different formats.

Available exporters:

5.1 Dictionary Exporter

```
class anytree.exporter.dictexporter.DictExporter(dictcls=<type 'dict'>, attriter=None,  
                                               childiter=<type 'list'>)
```

Bases: `object`

Tree to dictionary exporter.

Every node is converted to a dictionary with all instance attributes as key-value pairs. Child nodes are exported to the `children` attribute. A list of dictionaries.

Keyword Arguments

- `dictcls` – class used as dictionary. `Dictionary` displays by default.
- `attriter` – attribute iterator for sorting and/or filtering.
- `childiter` – child iterator for sorting and/or filtering.

```
>>> from pprint import pprint # just for nice printing  
>>> from anytree import AnyNode  
>>> from anytree.exporter import DictExporter  
>>> root = AnyNode(a="root")  
>>> s0 = AnyNode(a="sub0", parent=root)  
>>> s0a = AnyNode(a="sub0A", b="foo", parent=s0)  
>>> s0b = AnyNode(a="sub0B", parent=s0)  
>>> s1 = AnyNode(a="sub1", parent=root)
```

```
>>> exporter = DictExporter()
>>> pprint(exporter.export(root))  # order within dictionary might vary!
{'a': 'root',
 'children': [{ 'a': 'sub0',
    'children': [{ 'a': 'sub0A', 'b': 'foo'}, { 'a': 'sub0B'}]}, { 'a': 'sub1'}]}
```

Pythons dictionary *dict* does not preserve order. `collections.OrderedDict` does. In this case attributes can be ordered via *attriter*.

```
>>> from collections import OrderedDict
>>> exporter = DictExporter(dictcls=OrderedDict, attriter=sorted)
>>> pprint(exporter.export(root))
OrderedDict([('a', 'root'),
 ('children',
 [OrderedDict([('a', 'sub0'),
 ('children',
 [OrderedDict([('a', 'sub0A'), ('b', 'foo')]),
 OrderedDict([('a', 'sub0B')])])]),
 OrderedDict([('a', 'sub1')])])])
```

The attribute iterator *attriter* may be used for filtering too. For example, just dump attributes named *a*:

```
>>> exporter = DictExporter(attriter=lambda attrs: [(k, v) for k, v in attrs if k == "a"])
>>> pprint(exporter.export(root))
{'a': 'root',
 'children': [{ 'a': 'sub0', 'children': [{ 'a': 'sub0A'}, { 'a': 'sub0B'}]}, { 'a': 'sub1'}]}
```

The child iterator *childiter* can be used for sorting and filtering likewise:

```
>>> exporter = DictExporter(childiter=lambda children: [child for child in children if "0" in child.a])
>>> pprint(exporter.export(root))
{'a': 'root',
 'children': [{ 'a': 'sub0',
   'children': [{ 'a': 'sub0A', 'b': 'foo'}, { 'a': 'sub0B'}]}]}
```

export (node)

Export tree starting at *node*.

5.2 JSON Exporter

```
class anytree.exporter.jsonexporter.JsonExporter(dictexporter=None, **kwargs)
Bases: object
```

Tree to JSON exporter.

The tree is converted to a dictionary via *dictexporter* and exported to JSON.

Keyword Arguments

- **dictexporter** – Dictionary Exporter used (see *DictExporter*).
- **kwargs** – All other arguments are passed to `json.dump/json.dumps`. See documentation for reference.

```
>>> from anytree import AnyNode
>>> from anytree.exporter import JsonExporter
>>> root = AnyNode(a="root")
>>> s0 = AnyNode(a="sub0", parent=root)
>>> s0a = AnyNode(a="sub0A", b="foo", parent=s0)
>>> s0b = AnyNode(a="sub0B", parent=s0)
>>> s1 = AnyNode(a="sub1", parent=root)
```

```
>>> exporter = JsonExporter(indent=2, sort_keys=True)
>>> print(exporter.export(root))
{
  "a": "root",
  "children": [
    {
      "a": "sub0",
      "children": [
        {
          "a": "sub0A",
          "b": "foo"
        },
        {
          "a": "sub0B"
        }
      ]
    },
    {
      "a": "sub1"
    }
  ]
}
```

export (node)

Return JSON for tree starting at *node*.

write (node, filehandle)

Write JSON to *filehandle* starting at *node*.

5.3 Dot Exporter

For any details about the *dot* language, see [graphviz](#)

```
class anytree.exporter.dotexporter.DotExporter(node, graph='digraph', name='tree',
                                              options=None, indent=4, nodenamefunc=None,
                                              nodeattrfunc=None, edgeattrfunc=None,
                                              edgetypefunc=None)
```

Bases: `object`

Dot Language Exporter.

Parameters `node` (`Node`) – start node.

Keyword Arguments

- `graph` – DOT graph type.
- `name` – DOT graph name.

- **options** – list of options added to the graph.
- **indent** (*int*) – number of spaces for indent.
- **nodenamefunc** – Function to extract node name from *node* object. The function shall accept one *node* object as argument and return the name of it.
- **nodeattrfunc** – Function to decorate a node with attributes. The function shall accept one *node* object as argument and return the attributes.
- **edgeattrfunc** – Function to decorate a edge with attributes. The function shall accept two *node* objects as argument. The first the node and the second the child and return the attributes.
- **edgetypefunc** – Function to which gives the edge type. The function shall accept two *node* objects as argument. The first the node and the second the child and return the edge (i.e. ‘->’).

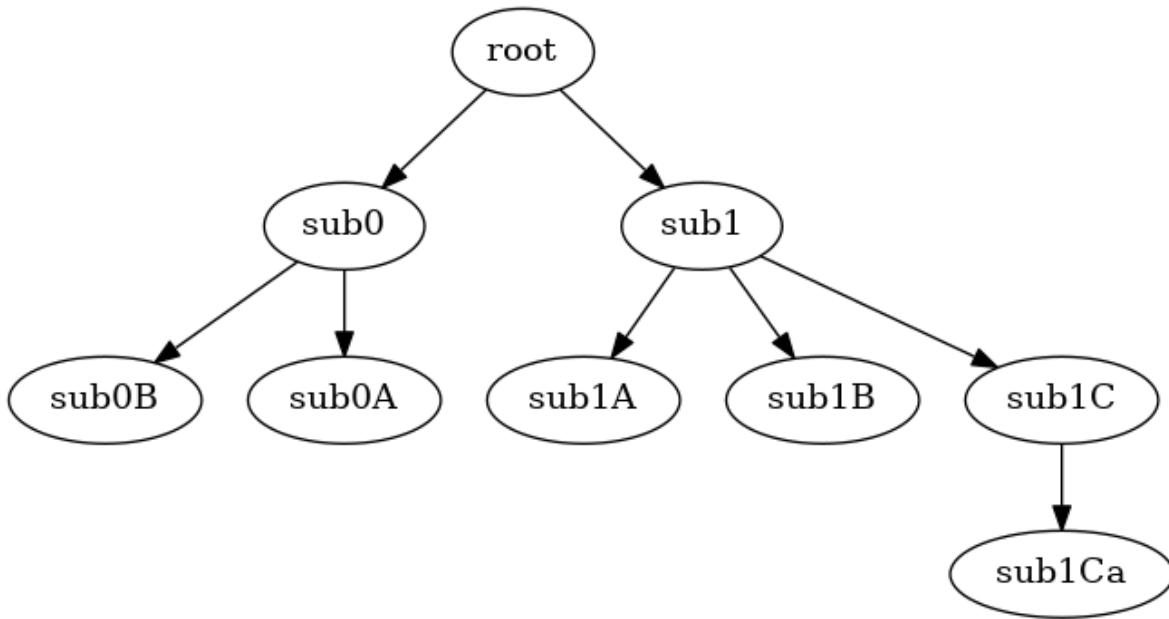
```
>>> from anytree import Node
>>> root = Node("root")
>>> s0 = Node("sub0", parent=root, edge=2)
>>> s0b = Node("sub0B", parent=s0, foo=4, edge=109)
>>> s0a = Node("sub0A", parent=s0, edge="")
>>> s1 = Node("sub1", parent=root, edge="")
>>> s1a = Node("sub1A", parent=s1, edge=7)
>>> s1b = Node("sub1B", parent=s1, edge=8)
>>> s1c = Node("sub1C", parent=s1, edge=22)
>>> s1ca = Node("sub1Ca", parent=s1c, edge=42)
```

Note: If the node names are not unique, see *UniqueDotExporter*.

A directed graph:

```
>>> from anytree.exporter import DotExporter
>>> for line in DotExporter(root):
...     print(line)
digraph tree {
    "root";
    "sub0";
    "sub0B";
    "sub0A";
    "sub1";
    "sub1A";
    "sub1B";
    "sub1C";
    "sub1Ca";
    "root" -> "sub0";
    "root" -> "sub1";
    "sub0" -> "sub0B";
    "sub0" -> "sub0A";
    "sub1" -> "sub1A";
    "sub1" -> "sub1B";
    "sub1" -> "sub1C";
    "sub1C" -> "sub1Ca";
}
```

The resulting graph:

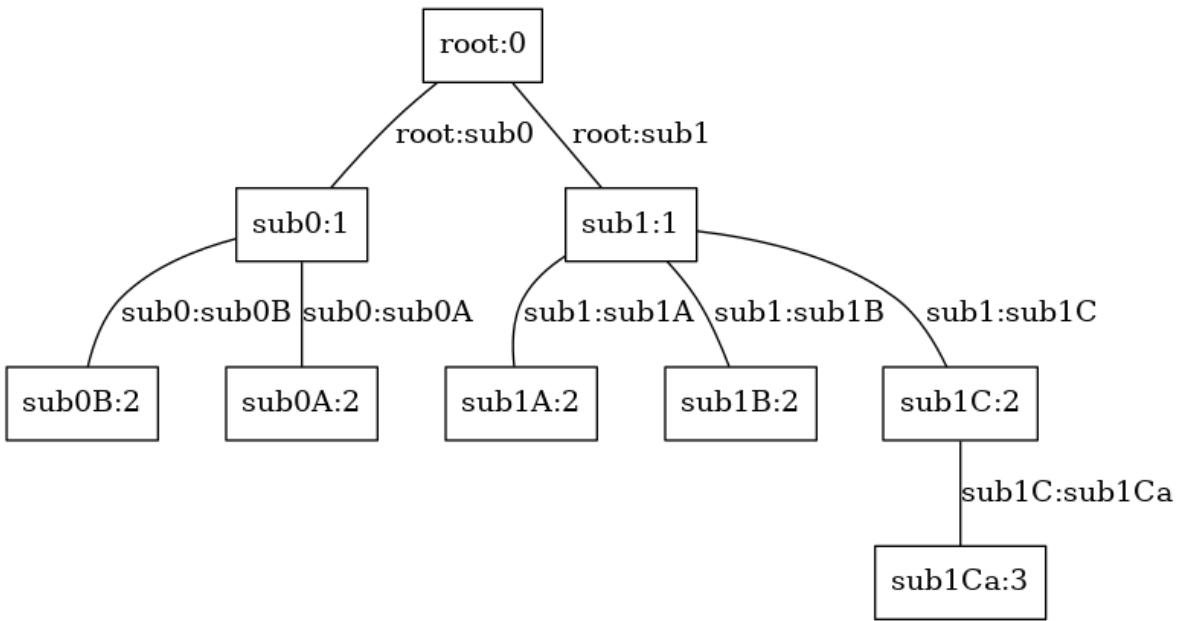


An undirected graph:

```

>>> def nodenamefunc(node):
...     return '%s:%s' % (node.name, node.depth)
>>> def edgeatrfunc(node, child):
...     return 'label="%s:%s"' % (node.name, child.name)
>>> def edgetypefunc(node, child):
...     return '--'
...     >>> from anytree.exporter import DotExporter
>>> for line in DotExporter(root, graph="graph",
...                         nodenamefunc=nodenamefunc,
...                         nodeatrfunc=lambda node: "shape=box",
...                         edgeatrfunc=edgeatrfunc,
...                         edgetypefunc=edgetypefunc):
...     print(line)
graph tree {
    "root:0" [shape=box];
    "sub0:1" [shape=box];
    "sub0B:2" [shape=box];
    "sub0A:2" [shape=box];
    "sub1:1" [shape=box];
    "sub1A:2" [shape=box];
    "sub1B:2" [shape=box];
    "sub1C:2" [shape=box];
    "sub1Ca:3" [shape=box];
    "root:0" -- "sub0:1" [label="root:sub0"];
    "root:0" -- "sub1:1" [label="root:sub1"];
    "sub0:1" -- "sub0B:2" [label="sub0:sub0B"];
    "sub0:1" -- "sub0A:2" [label="sub0:sub0A"];
    "sub1:1" -- "sub1A:2" [label="sub1:sub1A"];
    "sub1:1" -- "sub1B:2" [label="sub1:sub1B"];
    "sub1:1" -- "sub1C:2" [label="sub1:sub1C"];
    "sub1C:2" -- "sub1Ca:3" [label="sub1C:sub1Ca"];
}
  
```

The resulting graph:



To export custom node implementations or *AnyNode*, please provide a proper *nodenamefunc*:

```

>>> from anytree import AnyNode
>>> root = AnyNode(id="root")
>>> s0 = AnyNode(id="sub0", parent=root)
>>> s0b = AnyNode(id="s0b", parent=s0)
>>> s0a = AnyNode(id="s0a", parent=s0)
    
```

```

>>> from anytree.exporter import DotExporter
>>> for line in DotExporter(root, nodenamefunc=lambda n: n.id):
...     print(line)
digraph tree {
    "root";
    "sub0";
    "s0b";
    "s0a";
    "root" -> "sub0";
    "sub0" -> "s0b";
    "sub0" -> "s0a";
}
    
```

to_dotfile(filename)

Write graph to *filename*.

```

>>> from anytree import Node
>>> root = Node("root")
>>> s0 = Node("sub0", parent=root)
>>> s0b = Node("sub0B", parent=s0)
>>> s0a = Node("sub0A", parent=s0)
>>> s1 = Node("sub1", parent=root)
>>> s1a = Node("sub1A", parent=s1)
>>> s1b = Node("sub1B", parent=s1)
>>> s1c = Node("sub1C", parent=s1)
>>> s1ca = Node("sub1Ca", parent=s1c)
    
```

```
>>> from anytree.exporter import DotExporter
>>> DotExporter(root).to_dotfile("tree.dot")
```

The generated file should be handed over to the *dot* tool from the <http://www.graphviz.org/> package:

```
$ dot tree.dot -T png -o tree.png
```

to_picture(*filename*)

Write graph to a temporary file and invoke *dot*.

The output file type is automatically detected from the file suffix.

'graphviz' needs to be installed, before usage of this method.

static esc(*value*)

Escape Strings.

```
class anytree.exporter.dotexporter.UniqueDotExporter(node, graph='digraph',
                                                    name='tree', options=None,
                                                    indent=4, nodenamefunc=None,
                                                    nodeattrfunc=None, edgeattrfunc=None,
                                                    edgetypefunc=None)
```

Bases: *anytree.exporter.dotexporter.DotExporter*

Unqiue Dot Language Exporter.

Handle trees with random or conflicting node names gracefully.

Parameters **node** (*Node*) – start node.

Keyword Arguments

- **graph** – DOT graph type.
- **name** – DOT graph name.
- **options** – list of options added to the graph.
- **indent** (*int*) – number of spaces for indent.
- **nodenamefunc** – Function to extract node name from *node* object. The function shall accept one *node* object as argument and return the name of it.
- **nodeattrfunc** – Function to decorate a node with attributes. The function shall accept one *node* object as argument and return the attributes.
- **edgeattrfunc** – Function to decorate a edge with attributes. The function shall accept two *node* objects as argument. The first the node and the second the child and return the attributes.
- **edgetypefunc** – Function to which gives the edge type. The function shall accept two *node* objects as argument. The first the node and the second the child and return the edge (i.e. ' \rightarrow ').

```
>>> from anytree import Node
>>> root = Node("root")
>>> s0 = Node("sub0", parent=root)
>>> s0b = Node("s0", parent=s0)
>>> s0a = Node("s0", parent=s0)
>>> s1 = Node("sub1", parent=root)
```

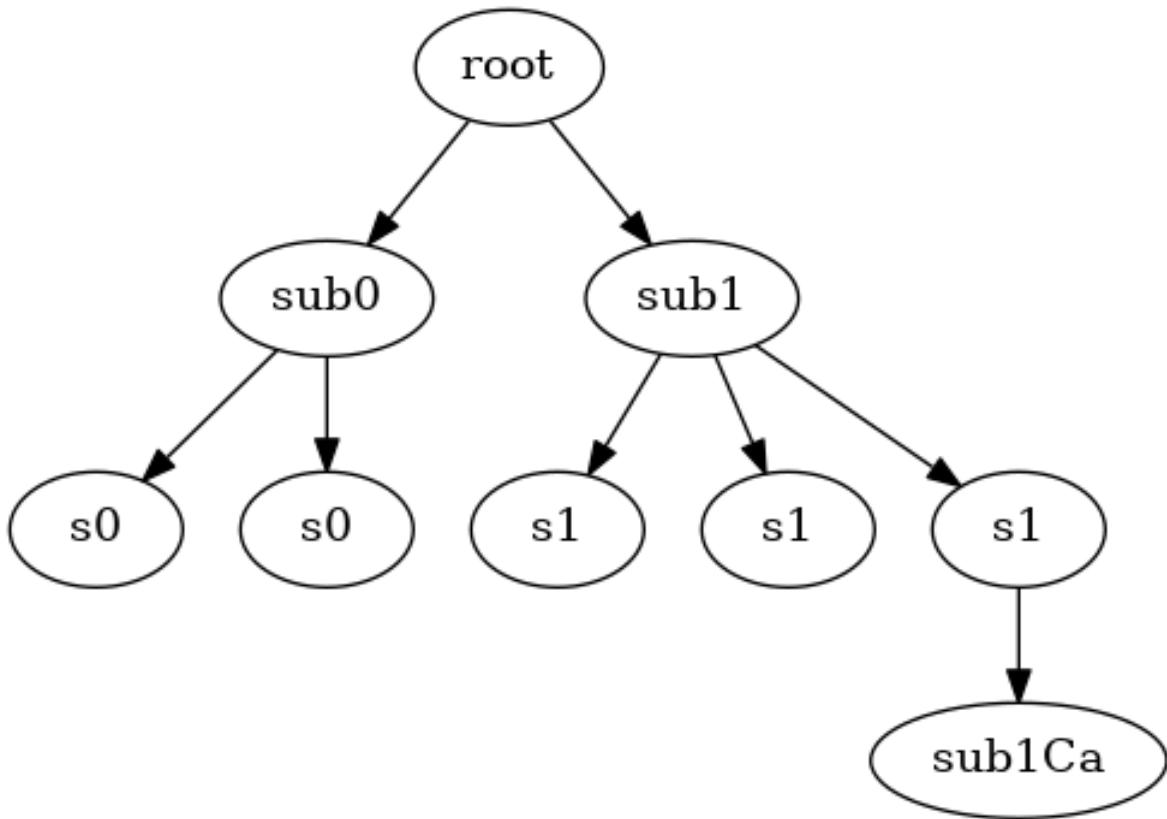
(continues on next page)

(continued from previous page)

```
>>> s1a = Node("s1", parent=s1)
>>> s1b = Node("s1", parent=s1)
>>> s1c = Node("s1", parent=s1)
>>> s1ca = Node("sub1Ca", parent=s1c)

>>> from anytree.exporter import UniqueDotExporter
>>> for line in UniqueDotExporter(root): # doctest: +SKIP
...     print(line)
digraph tree {
    "0x7f1bf2c9c510" [label="root"];
    "0x7f1bf2c9c5a0" [label="sub0"];
    "0x7f1bf2c9c630" [label="s0"];
    "0x7f1bf2c9c6c0" [label="s0"];
    "0x7f1bf2c9c750" [label="sub1"];
    "0x7f1bf2c9c7e0" [label="s1"];
    "0x7f1bf2c9c870" [label="s1"];
    "0x7f1bf2c9c900" [label="s1"];
    "0x7f1bf2c9c990" [label="sub1Ca"];
    "0x7f1bf2c9c510" -> "0x7f1bf2c9c5a0";
    "0x7f1bf2c9c510" -> "0x7f1bf2c9c750";
    "0x7f1bf2c9c5a0" -> "0x7f1bf2c9c630";
    "0x7f1bf2c9c5a0" -> "0x7f1bf2c9c6c0";
    "0x7f1bf2c9c750" -> "0x7f1bf2c9c7e0";
    "0x7f1bf2c9c750" -> "0x7f1bf2c9c870";
    "0x7f1bf2c9c750" -> "0x7f1bf2c9c900";
    "0x7f1bf2c9c900" -> "0x7f1bf2c9c990";
}
```

The resulting graph:



To export custom node implementations or `AnyNode`, please provide a proper `nodeattrfunc`:

```

>>> from anytree import AnyNode
>>> root = AnyNode(id="root")
>>> s0 = AnyNode(id="sub0", parent=root)
>>> s0b = AnyNode(id="s0", parent=s0)
>>> s0a = AnyNode(id="s0", parent=s0)

>>> from anytree.exporter import UniqueDotExporter
>>> for line in UniqueDotExporter(root, nodeattrfunc=lambda n: 'label=%s' % (n.
->id)): # doctest: +SKIP
...     print(line)
digraph tree {
    "0x7f5c70449af8" [label="root"];
    "0x7f5c70449bd0" [label="sub0"];
    "0x7f5c70449c60" [label="s0"];
    "0x7f5c70449cf0" [label="s0"];
    "0x7f5c70449af8" -> "0x7f5c70449bd0";
    "0x7f5c70449bd0" -> "0x7f5c70449c60";
    "0x7f5c70449bd0" -> "0x7f5c70449cf0";
}
  
```

Exporter missing? File a request here: [Issues](#).

CHAPTER 6

Tricks

6.1 Read-only Tree

Application: A read-only tree data structure, which denies modifications.

The `Node._pre_attach` and `Node._pre_detach` hookups can be used for blocking tree modifications. If they raise an `Exception`, the tree is not modified.

```
>>> from anytree import NodeMixin, RenderTree
```

The exception:

```
>>> class ReadOnlyError(RuntimeError):
...     pass
```

6.1.1 Permanent

The read-only attribute needs to be set after attaching to parent:

```
>>> class ReadOnlyNode(NodeMixin):
...
...     def __init__(self, foo, parent=None):
...         super(ReadOnlyNode, self).__init__()
...         self.foo = foo
...         self.__readonly = False
...         self.parent = parent
...         self.__readonly = True
...
...     def _pre_attach(self, parent):
...         if self.__readonly:
...             raise ReadOnlyError()
```

(continues on next page)

(continued from previous page)

```
...     def _pre_detach(self, parent):
...         raise ReadOnlyError()
```

An example tree:

```
>>> a = ReadOnlyNode("a")
>>> a0 = ReadOnlyNode("a0", parent=a)
>>> a1 = ReadOnlyNode("a1", parent=a)
>>> a1a = ReadOnlyNode("a1a", parent=a1)
>>> a2 = ReadOnlyNode("a2", parent=a)
>>> print(RenderTree(a).by_attr("foo"))
a
├── a0
├── a1
│   └── a1a
└── a2
```

Modifications raise an *ReadOnlyError*

```
>>> a0.parent = a2
Traceback (most recent call last):
...
ReadOnlyError
>>> a.children = [a1]
Traceback (most recent call last):
...
ReadOnlyError
```

The tree structure is untouched:

```
>>> print(RenderTree(a).by_attr("foo"))
a
├── a0
├── a1
│   └── a1a
└── a2
```

6.1.2 Temporary

To select the read-only mode temporarily, the root node should provide an attribute for all child nodes, set *after* construction.

```
>>> class ReadOnlyNode(NodeMixin):
...     def __init__(self, foo, parent=None):
...         super(ReadOnlyNode, self).__init__()
...         self.readonly = False
...         self.foo = foo
...         self.parent = parent
...     def _pre_attach(self, parent):
...         if self.root.readonly:
...             raise ReadOnlyError()
...     def _pre_detach(self, parent):
...         if self.root.readonly:
...             raise ReadOnlyError()
```

An example tree:

```
>>> a = ReadOnlyNode("a")
>>> a0 = ReadOnlyNode("a0", parent=a)
>>> a1 = ReadOnlyNode("a1", parent=a)
>>> a1a = ReadOnlyNode("a1a", parent=a1)
>>> a2 = ReadOnlyNode("a2", parent=a)
>>> print(RenderTree(a).by_attr("foo"))
a
└── a0
└── a1
    └── a1a
└── a2
```

Switch to read-only mode:

```
>>> a.readonly = True
```

```
>>> a0.parent = a2
Traceback (most recent call last):
...
ReadOnlyError
>>> a.children = [a1]
Traceback (most recent call last):
...
ReadOnlyError
```

Disable read-only mode:

```
>>> a.readonly = False
```

Modifications are allowed now:

```
>>> a0.parent = a2
>>> print(RenderTree(a).by_attr("foo"))
a
└── a1
    └── a1a
└── a2
    └── a0
```

6.2 YAML Import/Export

[YAML](#) (YAML Ain't Markup Language) is a human-readable data serialization language.

[PYYAML](#) implements importer and exporter in python. *Please install it, before continuing*

Note: anytree package does not depend on any external packages. It does **NOT** include PYYAML.

Warning: It is not safe to call `yaml.load` with any data received from an untrusted source! `yaml.load` is as powerful as `pickle.load` and so may call any Python function. The `yaml.safe_load` function limits the load functionality to built-in types.

6.2.1 Export

The *DictExporter* converts any tree to a dictionary, which can be handled by *yaml.dump*.

```
>>> import yaml
>>> from anytree import AnyNode
>>> from anytree.exporter import DictExporter
```

Example tree:

```
>>> root = AnyNode(a="root")
>>> s0 = AnyNode(a="sub0", parent=root)
>>> s0a = AnyNode(a="sub0A", b="foo", parent=s0)
>>> s0b = AnyNode(a="sub0B", parent=s0)
>>> s1 = AnyNode(a="sub1", parent=root)
```

Export to dictionary and convert to YAML:

```
>>> dct = DictExporter().export(root)
>>> print(yaml.dump(dct, default_flow_style=False))
a: root
children:
- a: sub0
  children:
  - a: sub0A
    b: foo
  - a: sub0B
- a: sub1
<BLANKLINE>
```

DictExporter controls the content. *yaml.dump* controls the YAML related stuff.

To dump to a file, use an file object as second argument:

```
>>> with open("/path/to/file", "w") as file: # doctest: +SKIP
...     yaml.dump(data, file)
```

6.2.2 Import

The *yaml.load* function reads YAML data — a dictionary, which *DictImporter* converts to a tree.

```
>>> import yaml
>>> from anytree.importer import DictImporter
>>> from pprint import pprint # just for nice printing
>>> from anytree import RenderTree # just for nice printing
```

Example data:

```
>>> data = """
... a: root
... children:
... - a: sub0
...   children:
...   - a: sub0A
...     b: foo
...   - a: sub0B
```

(continues on next page)

(continued from previous page)

```
... - a: sub1
...
"""


```

Import to dictionary and convert to tree:

```
>>> dct = yaml.load(data)
>>> pprint(dct)
{'a': 'root',
 'children': [{}{'a': 'sub0',
                 'children': [{}{'a': 'sub0A', 'b': 'foo'}, {'a': 'sub0B'}]}, {}{'a': 'sub1'}]}
>>> root = DictImporter().import_(dct)
>>> print(RenderTree(root))
AnyNode(a='root')
└── AnyNode(a='sub0')
    ├── AnyNode(a='sub0A', b='foo')
    └── AnyNode(a='sub0B')
└── AnyNode(a='sub1')
```

6.3 Multidimensional Trees

Application: Tree nodes should be hooked-up in multiple trees.

An anytree node is only able to be part of **one** tree, not multiple. The following example shows how to handle this.

Example: 4 objects *A*, *B*, *C* and *D* shall be part of the trees *X* and *Y*.

The objects *A*, *B*, *C* and *D* are instances of a class *Item*. It is *not* a tree node. It just contains references *x* and *y* to the node representations in the corresponding trees.

```
>>> class Item:
...     def __init__(self, name):
...         self.name = name
...         self.x = None
...         self.y = None
...     def __repr__(self):
...         return "Item(%r)" % (self.name)
>>> a = Item('A')
>>> b = Item('B')
>>> c = Item('C')
>>> d = Item('D')
```

The tree nodes just contain the reference to *item* and take care of the proper reference, by using the attach/detach protocol.

```
>>> from anytree import NodeMixin, RenderTree
>>> class NodeX(NodeMixin):
...     def __init__(self, item, parent=None):
...         self.item = item
...         self.parent = parent
...     def _pre_detach(self, parent):
...         self.item.x = None
...     def _pre_attach(self, parent):
...         self.item.x = self
>>> class NodeY(NodeMixin):
```

(continues on next page)

(continued from previous page)

```
...     def __init__(self, item, parent=None):
...
...         self.item = item
...
...         self.parent = parent
...
...     def _pre_detach(self, parent):
...         self.item.y = None
...
...     def _pre_attach(self, parent):
...         self.item.y = self
```

Tree generation is simple:

```
>>> # X
>>> xa = NodeX(a)
>>> xb = NodeX(b, parent=xa)
>>> xc = NodeX(c, parent=xa)
>>> xd = NodeX(d, parent=xc)
>>> # Y
>>> yd = NodeY(d)
>>> yc = NodeY(c, parent=yd)
>>> yb = NodeY(b, parent=yd)
>>> ya = NodeY(a, parent=yb)
```

All tree functions as rendering and exporting can be used as usual:

```
>>> for row in RenderTree(xa):
...     print("%s%s" % (row.pre, row.node.item))
Item('A')
└── Item('B')
    ├── Item('C')
    └── Item('D')
```

```
>>> for row in RenderTree(yd):
...     print("%s%s" % (row.pre, row.node.item))
Item('D')
└── Item('C')
    ├── Item('B')
    └── Item('A')
```

CHAPTER 7

Getting started

Usage is simple.

Construction

```
>>> from anytree import Node, RenderTree
>>> udo = Node("Udo")
>>> marc = Node("Marc", parent=udo)
>>> lian = Node("Lian", parent=marc)
>>> dan = Node("Dan", parent=udo)
>>> jet = Node("Jet", parent=dan)
>>> jan = Node("Jan", parent=dan)
>>> joe = Node("Joe", parent=dan)
```

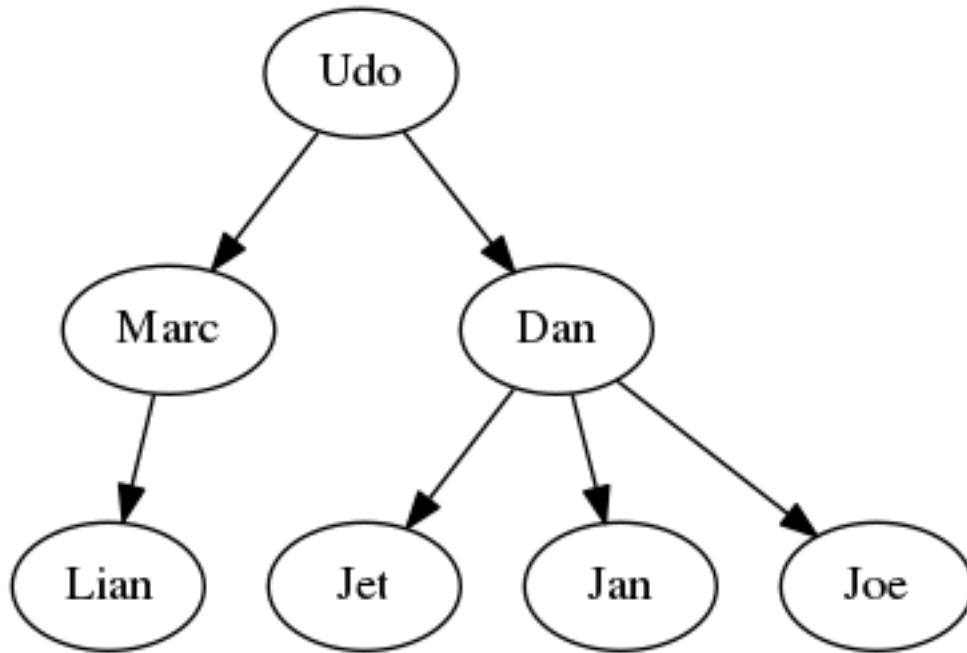
Node

```
>>> print(udo)
Node('/Udo')
>>> print(joe)
Node('/Udo/Dan/Joe')
```

Tree

```
>>> for pre, fill, node in RenderTree(udo):
...     print("%s%s" % (pre, node.name))
Udo
└── Marc
    └── Lian
        └── Dan
            ├── Jet
            ├── Jan
            └── Joe
```

```
>>> from anytree.exporter import DotExporter
>>> # graphviz needs to be installed for the next line!
>>> DotExporter(udo).to_picture("udo.png")
```



Manipulation

A second tree:

```

>>> mary = Node("Mary")
>>> urs = Node("Urs", parent=mary)
>>> chris = Node("Chris", parent=mary)
>>> marta = Node("Marta", parent=mary)
>>> print(RenderTree(mary))
Node('/Mary')
└── Node('/Mary/Urs')
└── Node('/Mary/Chris')
└── Node('/Mary/Marta')
  
```

Append:

```

>>> udo.parent = mary
>>> print(RenderTree(mary))
Node('/Mary')
└── Node('/Mary/Urs')
└── Node('/Mary/Chris')
└── Node('/Mary/Marta')
└── Node('/Mary/Udo')
    ├── Node('/Mary/Udo/Marc')
    │   └── Node('/Mary/Udo/Marc/Lian')
    └── Node('/Mary/Udo/Dan')
        ├── Node('/Mary/Udo/Dan/Jet')
        ├── Node('/Mary/Udo/Dan/Jan')
        └── Node('/Mary/Udo/Dan/Joe')
  
```

Subtree rendering:

```

>>> print(RenderTree(marc))
Node('/Mary/Udo/Marc')
└── Node('/Mary/Udo/Marc/Lian')
  
```

Cut:

```
>>> dan.parent = None
>>> print(RenderTree(dan))
Node('/Dan')
└── Node('/Dan/Jet')
└── Node('/Dan/Jan')
└── Node('/Dan/Joe')
```

Extending any python class to become a tree node

```
>>> from anytree import NodeMixin, RenderTree
>>> class MyBaseClass(object): # Just an example of a base class
...     foo = 4
>>> class MyClass(MyBaseClass, NodeMixin): # Add Node feature
...     def __init__(self, name, length, width, parent=None, children=None):
...         super(MyClass, self).__init__()
...         self.name = name
...         self.length = length
...         self.width = width
...         self.parent = parent
...         if children:
...             self.children = children
```

Just set the *parent* attribute to reflect the tree relation:

```
>>> my0 = MyClass('my0', 0, 0)
>>> my1 = MyClass('my1', 1, 0, parent=my0)
>>> my2 = MyClass('my2', 0, 2, parent=my0)
```

```
>>> for pre, fill, node in RenderTree(my0):
...     treestr = u"%s%s" % (pre, node.name)
...     print(treestr.ljust(8), node.length, node.width)
my0      0 0
└── my1  1 0
    └── my2 0 2
```

The *children* can be used likewise:

```
>>> my0 = MyClass('my0', 0, 0, children=[
...     MyClass('my1', 1, 0),
...     MyClass('my2', 0, 2),
... ])
```

```
>>> for pre, fill, node in RenderTree(my0):
...     treestr = u"%s%s" % (pre, node.name)
...     print(treestr.ljust(8), node.length, node.width)
my0      0 0
└── my1  1 0
    └── my2 0 2
```

Python Module Index

a

anytree.cachedsearch, 30
anytree.exporter.dictexporter, 37
anytree.exporter.dotexporter, 39
anytree.exporter.jsonexporter, 38
anytree.importer.dictimporter, 35
anytree.importer.jsonimporter, 36
anytree.iterators, 17
anytree.iterators.levelordergroupiter,
 20
anytree.iterators.levelorderiter, 19
anytree.iterators.postorderiter, 18
anytree.iterators.preorderiter, 17
anytree.iterators.zigzaggroupiter, 21
anytree.node, 9
anytree.node.anynode, 9
anytree.node.exceptions, 17
anytree.node.node, 10
anytree.node.nodemixin, 11
anytree.render, 22
anytree.resolver, 30
anytree.search, 26
anytree.util, 33
anytree.walker, 32

Index

A

AbstractStyle (*class in anytree.render*), 22
ancestors (*anytree.node.nodemixin.NodeMixin attribute*), 14
ancestors (*anytree.node.nodemixin.NodeMixin attribute*), 15
AnyNode (*class in anytree.node.anynode*), 9
anytree.cachedsearch (*module*), 30
anytree.exporter.dictexporter (*module*), 37
anytree.exporter.dotexporter (*module*), 39
anytree.exporter.jsonexporter (*module*), 38
anytree.importer.dictimporter (*module*), 35
anytree.importer.jsonimporter (*module*), 36
anytree.iterators (*module*), 17
anytree.iterators.levelordergroupiter (*module*), 20
anytree.iterators.levelorderiter (*module*), 19
anytree.iterators.postorderiter (*module*), 18
anytree.iterators.preorderiter (*module*), 17
anytree.iterators.zigzaggroupiter (*module*), 21
anytree.node (*module*), 9
anytree.node.anynode (*module*), 9
anytree.node.exceptions (*module*), 17
anytree.node.node (*module*), 10
anytree.node.nodemixin (*module*), 11
anytree.render (*module*), 22
anytree.resolver (*module*), 30
anytree.search (*module*), 26
anytree.util (*module*), 33
anytree.walker (*module*), 32
AsciiStyle (*class in anytree.render*), 22

B

by_attr () (*anytree.render.RenderTree method*), 25

C

children (*anytree.node.nodemixin.NodeMixin attribute*), 13
ChildResolverError, 32
commonancestors () (*in module anytree.util*), 33
ContRoundStyle (*class in anytree.render*), 23
ContStyle (*class in anytree.render*), 23
CountError, 29

D

depth (*anytree.node.nodemixin.NodeMixin attribute*), 17
descendants (*anytree.node.nodemixin.NodeMixin attribute*), 15
DictExporter (*class in anytree.exporter.dictexporter*), 37
DictImporter (*class in anytree.importer.dictimporter*), 35
DotExporter (*class in anytree.exporter.dotexporter*), 39
DoubleStyle (*class in anytree.render*), 23

E

empty (*anytree.render.AbstractStyle attribute*), 22
esc () (*anytree.exporter.dotexporter.DotExporter static method*), 43
export () (*anytree.exporter.dictexporter.DictExporter method*), 38
export () (*anytree.exporter.jsonexporter.JsonExporter method*), 39

F

fill (*anytree.render.Row attribute*), 22
find () (*in module anytree.cachedsearch*), 30
find () (*in module anytree.search*), 28
find_by_attr () (*in module anytree.cachedsearch*), 30
find_by_attr () (*in module anytree.search*), 29
findall () (*in module anytree.cachedsearch*), 30

findall () (in module `anytree.search`), 26
 .findall_by_attr () (in module `anytree.cachedsearch`), 30
 .findall_by_attr () (in module `anytree.search`), 27

G

get () (`anytree.resolver.Resolver` method), 30
 .glob () (`anytree.resolver.Resolver` method), 31

H

height (`anytree.node.nodemixin.NodeMixin` attribute), 16

I

import_ () (`anytree.importer.dictimporter.DictImporter` method), 35
 import_ () (`anytree.importer.jsonimporter.JsonImporter` method), 36
 .is_leaf (`anytree.node.nodemixin.NodeMixin` attribute), 16
 .is_root (`anytree.node.nodemixin.NodeMixin` attribute), 16
 .is_wildcard () (`anytree.resolver.Resolver` static method), 32
 iter_path_reverse () (`anytree.node.nodemixin.NodeMixin` method), 14

J

`JsonExporter` (class in `anytree.exporter.jsonexporter`), 38
`JsonImporter` (class in `anytree.importer.jsonimporter`), 36

L

.leaves (`anytree.node.nodemixin.NodeMixin` attribute), 16
 leftsibling () (in module `anytree.util`), 34
 LevelOrderGroupIter (class in `anytree.iterators.levelordergroupiter`), 20
 LevelOrderIter (class in `anytree.iterators.levelorderiter`), 19
 LoopError, 17

N

node (`anytree.render.Row` attribute), 22
 Node (class in `anytree.node.node`), 10
 NodeMixin (class in `anytree.node.nodemixin`), 11

P

parent (`anytree.node.nodemixin.NodeMixin` attribute), 12
 path (`anytree.node.nodemixin.NodeMixin` attribute), 14

PostOrderIter (class in `anytree.iterators.postorderiter`), 18
 pre (`anytree.render.Row` attribute), 22
 PreOrderIter (class in `anytree.iterators.preorderiter`), 17

R

read () (`anytree.importer.jsonimporter.JsonImporter` method), 36
 RenderTree (class in `anytree.render`), 24
 Resolver (class in `anytree.resolver`), 30
 ResolverError, 32
 rightsibling () (in module `anytree.util`), 34
 root (`anytree.node.nodemixin.NodeMixin` attribute), 15
 Row (class in `anytree.render`), 22

S

separator (`anytree.node.nodemixin.NodeMixin` attribute), 11
 siblings (`anytree.node.nodemixin.NodeMixin` attribute), 15

T

to_dotfile () (`anytree.exporter.dotexporter.DotExporter` method), 42
 to_picture () (`anytree.exporter.dotexporter.DotExporter` method), 43
 TreeError, 17

U

UniqueDotExporter (class in `anytree.exporter.dotexporter`), 43

W

walk () (`anytree.walker.Walker` method), 32
 Walker (class in `anytree.walker`), 32
 WalkError, 33
 write () (`anytree.exporter.jsonexporter.JsonExporter` method), 39

Z

ZigZagGroupIter (class in `anytree.iterators.zigzaggroupiter`), 21