Python Tree Data

Release 2.2.2

c0fec0de

Nov 06, 2017

Contents

Simple, lightweight and extensible Tree data structure.

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
- `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.

```
>>> 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 an *children* attribute with a tuple of all children:

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

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
```

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* functions, 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')
```

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:

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

CHAPTER 3

API

3.1 Node Classes

Node Classes.

- *Node*: a simple tree node
- *NodeMixin*: extends any python class to a tree node.

class anytree.node.**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.

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

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.

```
>>> dan = Node("Dan")
>>> jet = Node("Jet", parent=dan)
>>> jan = Node("Jan", parent=dan)
>>> joe = Node("Joe", parent=dan)
>>> dan.children
(Node('/Dan/Jet'), Node('/Dan/Jan'), Node('/Dan/Joe'))
```

path

Path of this *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'))
```

ancestors
All parent nodes and their parent nodes.

```
>>> 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'))
```

anchestors
All parent nodes and their parent nodes - see *ancestors*.

The attribute *anchestors* is just a typo of *ancestors*. Please use *ancestors*. This attribute will be removed in the 2.0.0 release.

descendants
All child nodes and all their child nodes.

```
>>> 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.

```
>>> 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.

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

is_leaf

Node has no childrean (External 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.

```
>>> 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*.

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

depth

Number of edges to the root *Node*.

```
>>> udo = Node("Udo")
>>> marc = Node("Marc", parent=udo)
>>> lian = Node("Lian", parent=marc)
>>> udo.depth
0
>>> marc.depth
```

```
1
>>> lian.depth
2
```

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

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

```
>>> 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')
```

name
Name.

exception anytree.node.**LoopError**
Bases: exceptions.RuntimeError

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.**AbstractIter**(*node*, *filter_=None*, *stop=None*, *maxlevel=None*)
Bases: six.Iterator

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 decending in the node hierarchy.

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

Bases: *anytree.iterators.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
>>> 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 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 decending in the node hierarchy.

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

Bases: *anytree.iterators.AbstractIter*

Iterate over tree applying post-order strategy starting at *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)
>>> 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 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 decending in the node hierarchy.

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

Bases: *anytree.iterators.AbstractIter*

Iterate over tree applying level-order strategy starting at *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)
>>> 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'))]
['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 decending in the node hierarchy.

class anytree.iterators.**LevelOrderGroupIter**(*node*, *filter_=None*, *stop=None*, *maxlevel=None*)

Bases: *anytree.iterators.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
>>> 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 decending in the node hierarchy.

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

Bases: *anytree.iterators.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 decending in the node hierarchy.

3.3 Tree Rendering

Tree Rendering.

- ***RenderTree*** **using the following styles:**
  - *AsciiStyle*
  - *ContStyle*
  - *ContRoundStyle*
  - *DoubleStyle*

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
|   - sub0A
- 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
|   - a
|       b
- Z
```

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

```
>>> for pre, _, node in RenderTree(root, childiter=reversed):
...     print("%s%s" % (pre, 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 pre, _, node in RenderTree(root, childiter=mysort):
...     print("%s%s" % (pre, 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
```

by_attr(*attrname='name'*)
Return rendered tree with node attribute *attrname*.

3.4 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):
  ...
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/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* 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*.

3.5 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)
>>> 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.

CHAPTER 4

Export to DOT

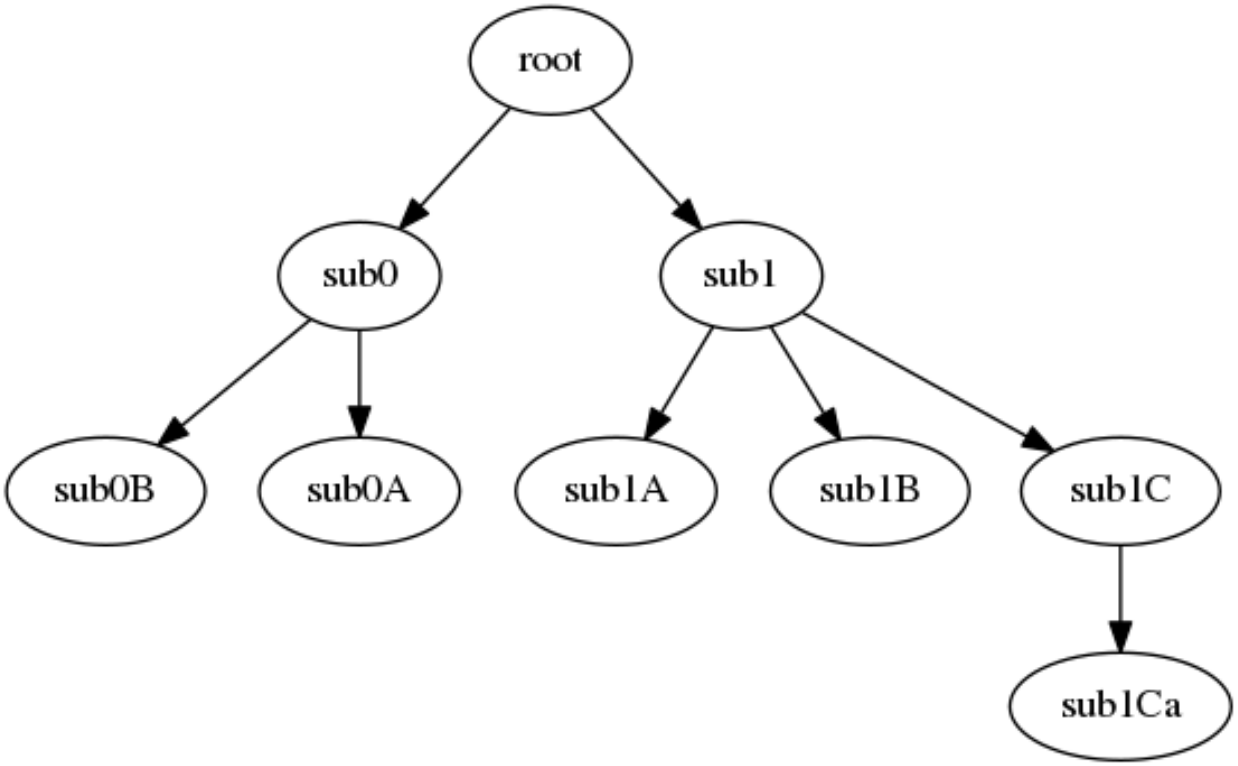
Any `anytree` graph can be converted to a graphviz graph.

This tree:

```
>>> 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)
```

Can be rendered to a tree by *RenderTreeGraph*:

```
>>> from anytree.dotexport import RenderTreeGraph
>>> RenderTreeGraph(root).to_picture("tree.png")
```



class anytree.dotexport.**RenderTreeGraph**(*node*, *graph='digraph'*, *name='tree'*, *options=None*, *indent=4*, *nodenamefunc=None*, *nodeattrfunc=None*, *edgeattrfunc=None*, *edgetypefunc=None*)

Bases: anytree.dotexport._Render

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)
```

A directed graph:

```
>>> for line in RenderTreeGraph(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";
}
```

An undirected graph:

```
>>> def nodenamefunc(node):
...     return '%s:%s' % (node.name, node.depth)
>>> def edgeattrfunc(node, child):
...     return 'label="%s:%s"' % (node.name, child.name)
>>> def edgetypefunc(node, child):
...     return '--'
>>> for line in RenderTreeGraph(root, graph="graph",
...                             nodenamefunc=nodenamefunc,
...                             nodeattrfunc=lambda node: "shape=box",
...                             edgeattrfunc=edgeattrfunc,
...                             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"];
}
```

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)
```

```
>>> RenderTreeGraph(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.

CHAPTER 5

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)
```

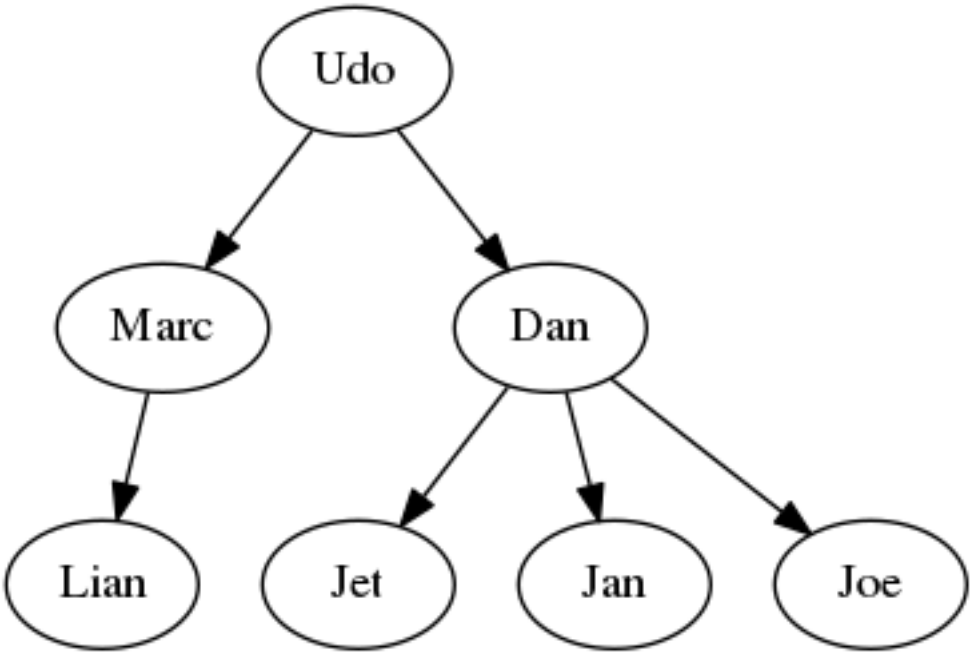
Node

```
>>> 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.dotexport import RenderTreeGraph
>>> # graphviz needs to be installed for the next line!
>>> RenderTreeGraph(root).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')
```

Python Module Index

a

Index

Z