Python Tree Data

Release 1.0.1

c0fec0de

Mar 13, 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

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)
```

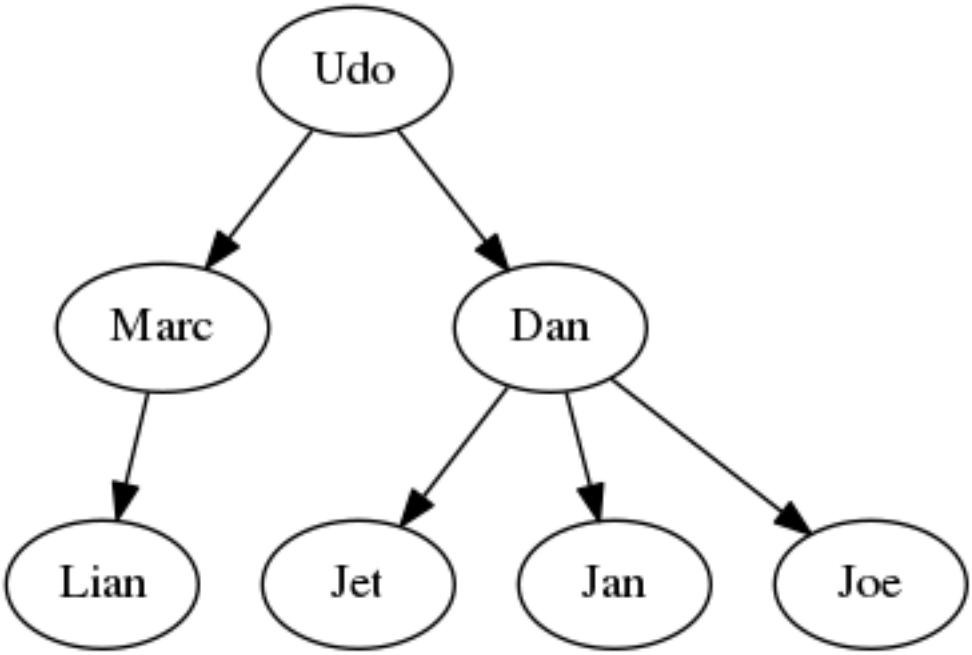
Node

```
>>> 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')
```

CHAPTER 3

API

Powerful and Lightweight Python Tree Data Structure.

Overview

The *anytree* API is splitted into the following parts:

- **Node classes:**
  - *Node*: a simple tree node
  - *NodeMixin*: extends any python class to a tree node.
- **Tree Traversal strategies:**
  - *PreOrderIter*: iterate over tree using pre-order strategy
  - *PostOrderIter*: iterate over tree using post-order strategy
- **Tree Rendering:**
  - ***RenderTree*** **using the following styles:**
    - *AsciiStyle*
    - *ContStyle*
    - *ContRoundStyle*
    - *DoubleStyle*

Classes

class anytree.**NodeMixin**
Bases: object

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.

```
>>> 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.

```
>>> 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')
```

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

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'))
```

anchestors
All parent nodes and their parent nodes.

```
>>> udo = Node("Udo")
>>> marc = Node("Marc", parent=udo)
>>> lian = Node("Lian", parent=marc)
>>> udo.anchestors
()
>>> marc.anchestors
(Node('Udo'),)
>>> lian.anchestors
(Node('Udo'), Node('Udo/Marc'))
```

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)
>>> udo.descendants
(Node('Udo/Marc'), Node('Udo/Marc/Lian'), Node('Udo/Marc/Loui'))
>>> marc.descendants
(Node('Udo/Marc/Lian'), Node('Udo/Marc/Loui'))
>>> lian.descendants
()
```

root
Tree Root Node.

```
>>> udo = Node("Udo")
>>> marc = Node("Marc", parent=udo)
>>> lian = Node("Lian", parent=marc)
>>> udo.root is None
True
>>> 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**(*name*, *parent=None*, ***kwargs*)

Bases: *anytree.NodeMixin*, object

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

```
>>> 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.

class anytree.**PreOrderIter**(*node*)

Bases: object

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

```
>>> 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)
```

```
>>> [node.name for node in PreOrderIter(f)]
['f', 'b', 'a', 'd', 'c', 'e', 'g', 'i', 'h']
```

class anytree.**PostOrderIter**(*node*)

Bases: object

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

```
>>> 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)
```

```
>>> [node.name for node in PostOrderIter(f)]
['a', 'c', 'e', 'd', 'b', 'h', 'i', 'g', 'f']
```

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

Bases: object

Tree Render Style.

Args:

vertical: Sign for vertical line.

cont: Chars for a continued branch.

end: Chars for the last branch.

empty

Empty string as placeholder.

class anytree.**AsciiStyle**

Bases: *anytree.AbstractStyle*

Ascii style.

```
>>> 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.**ContStyle**

Bases: *anytree.AbstractStyle*

Continued style, without gaps.

```
>>> 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.**ContRoundStyle**

Bases: *anytree.AbstractStyle*

Continued style, without gaps, round edges.

```
>>> 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.**DoubleStyle**

Bases: *anytree.AbstractStyle*

Double line style, without gaps.

```
>>> 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.**RenderTree**(*node*, *style=ContStyle()*, *childiter=<type 'list'>*)

Bases: object

Render tree starting at *node*.

Keyword Args: 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.

```
>>> 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
```

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

Tree contains infinite loop.

CHAPTER 4

Export to DOT

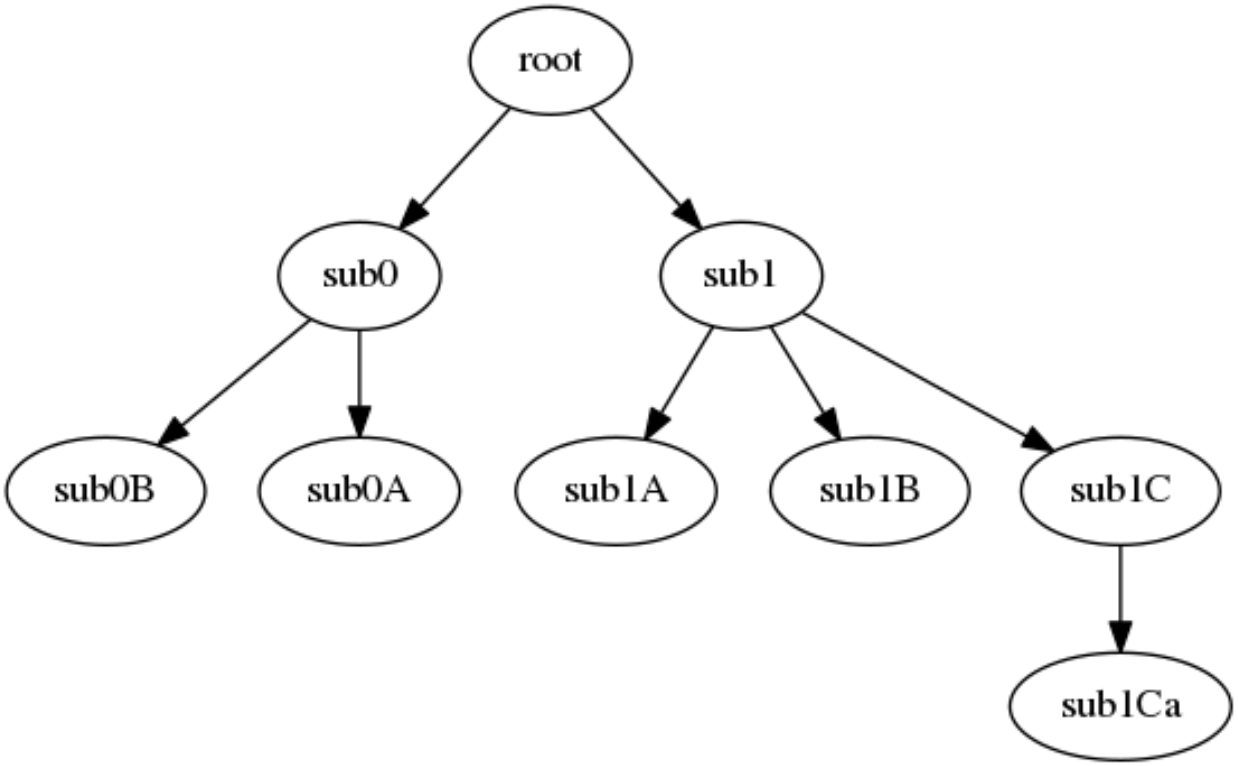
Any *anytree* graph can be converted to a graphviz graph.

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*)

Bases: anytree.dotexport._Render

Dot Language Exporter.

Args: node (Node): start node.

Keyword Args: 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.

```
>>> 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)
```

```
>>> 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";
}
```

```
>>> def nodenamefunc(node):
...     return '%s:%s' % (node.name, node.depth)
>>> def edgeattrfunc(node, child):
...     return 'label="%s:%s"' % (node.name, child.name)
>>> for line in RenderTreeGraph(root, options=["rankdir=LR;"],
...                             nodenamefunc=nodenamefunc,
...                             nodeattrfunc=lambda node: "shape=box",
...                             edgeattrfunc=edgeattrfunc):
...     print(line)
digraph tree {
    rankdir=LR;
    "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.

Python Module Index

a

Index