Baron Documentation

Release 0.6

Laurent Peuch

Apr 22, 2018

Contents

1 Introduction **1**

2 Github (code, bug tracker, etc.) **3**

3 Installation **5**

4 RedBaron **7**

5 Basic usage **9**

6 Table of content **11**
- 6.1 Basic Usage . . . 11
- 6.2 Advanced Usage . . . 14
- 6.3 Rendering the FST . . . 18

7 Indices and tables **27**

CHAPTER 1

Introduction

Baron is a Full Syntax Tree (FST) for Python. It represents source code as a structured tree, easily parsable by a computer. By opposition to an Abstract Syntax Tree (AST) which drops syntax information in the process of its creation (like empty lines, comments, formatting), a FST keeps everything and guarantees the operation `fst_to_code(code_to_fst(source_code)) == source_code`.

If you want to understand why this is important, read this: https://github.com/PyCQA/baron#why-is-this-important

CHAPTER 2

Github (code, bug tracker, etc.)

https://github.com/PyCQA/baron

CHAPTER 3

Installation

```
pip install baron
```

CHAPTER 4

RedBaron

There is a good chance that you'll want to use RedBaron instead of using Baron directly. Think of Baron as the "bytecode of python source code" and RedBaron as some sort of usable layer on top of it, a bit like dom/jQuery or html/Beautifulsoup.

CHAPTER 5

Basic usage

```
In [1]: from baron import parse, dumps

In [2]: source_code = "a = 1"

In [3]: fst = parse(source_code)

In [4]: fst
Out[4]:
[{'first_formatting': [{'type': 'space', 'value': ' '}],
  'operator': '',
  'second_formatting': [{'type': 'space', 'value': ' '}],
  'target': {'type': 'name', 'value': 'a'},
  'type': 'assignment',
  'value': {'section': 'number', 'type': 'int', 'value': '1'}}]

In [5]: generated_source_code = dumps(fst)

In [6]: generated_source_code
Out[6]: 'a = 1'

In [7]: source_code == generated_source_code
Out[7]: True
```

CHAPTER 6

Table of content

6.1 Basic Usage

Baron provides two main functions:

- `parse` to transform a string into Baron's FST;
- `dumps` to transform the FST back into a string.

```
In [1]: from baron import parse, dumps

In [2]: source_code = "def f(x = 1):\n    return x\n"

In [3]: fst = parse(source_code)

In [4]: generated_source_code = dumps(fst)

In [5]: generated_source_code
Out[5]: 'def f(x = 1):\n    return x\n'

In [6]: source_code == generated_source_code
Out[6]: True
```

Like said in the introduction, the FST keeps the formatting unlike ASTs. Here the following 3 codes are equivalent but their formatting is different. Baron keeps the difference so when dumping back the FST, all the formatting is respected:

```
In [7]: dumps(parse("a = 1"))
Out[7]: 'a = 1'

In [8]: dumps(parse("a=1"))
Out[8]: 'a=1'

In [9]: dumps(parse("a   =   1"))
Out[9]: 'a   =   1'
```

6.1.1 Helpers

Baron also provides 3 helper functions *show*, *show_file* and *show_node* to explore the FST (in iPython for example). Those functions will print a formatted version of the FST so you can play with it to explore the FST and have an idea of what you are playing with.

Show

`show` is used directly on a string:

```
In [10]: from baron.helpers import show

In [11]: show("a = 1")
[
    {
        "first_formatting": [
            {
                "type": "space",
                "value": " "
            }
        ],
        "target": {
            "type": "name",
            "value": "a"
        },
        "value": {
            "section": "number",
            "type": "int",
            "value": "1"
        },
        "second_formatting": [
            {
                "type": "space",
                "value": " "
            }
        ],
        "operator": "",
        "type": "assignment"
    }
]

In [12]: show("a +=  b")
[
    {
        "first_formatting": [
            {
                "type": "space",
                "value": " "
            }
        ],
        "target": {
            "type": "name",
            "value": "a"
        },
        "value": {
            "type": "name",
            "value": "b"
```

```
        },
        "second_formatting": [
            {
                "type": "space",
                "value": "  "
            }
        ],
        "operator": "+",
        "type": "assignment"
    }
]
```

Show_file

`show_file` is used on a file path:

```
from baron.helpers import show_file

show_file("/path/to/a/file")
```

Show_node

`show_node` is used on an already parsed string:

```
In [13]: from baron.helpers import show_node

In [14]: fst = parse("a = 1")

In [15]: show_node(fst)
[
    {
        "first_formatting": [
            {
                "type": "space",
                "value": " "
            }
        ],
        "target": {
            "type": "name",
            "value": "a"
        },
        "value": {
            "section": "number",
            "type": "int",
            "value": "1"
        },
        "second_formatting": [
            {
                "type": "space",
                "value": " "
            }
        ],
        "operator": "",
        "type": "assignment"
```

```
    }
]
```

Under the hood, the FST is serialized into JSON so the helpers are simply encapsulating JSON pretty printers.

6.2 Advanced Usage

The topics presented here are less often needed but are still very useful.

6.2.1 Locate a Node

Since Baron produces a tree, a path is sufficient to locate univocally a node in the tree. A common task where a path is involved is when translating a position in a file (a line and a column) into a node of the FST.

Baron provides 2 helper functions for that:

- `position_to_node(fst, line, column)`
- `position_to_path(fst, line, column)`

Both take a FST tree as first argument, then the line number and the column number. Line and column numbers **start at 1**, like in a text editor.

`position_to_node` returns an FST node. This is okay if you only want to know which node it is but not enough to locate the node in the tree. Indeed, there can be mutiple identical nodes within the tree.

That's where `position_to_path` is useful. It returns a list of int and strings which represent either the key to take in a Node or the index in a ListNode. For example: `["target", "value", 0]`)

Let's first see the difference between the two functions:

```
In [1]: from baron import parse

In [2]: from baron.path import position_to_node, position_to_path

In [3]: from baron.helpers import show_node

In [4]: some_code = """from baron import parse\nfrom baron.helpers import show_
↪node\nfst = parse("a = 1")\nshow_node(fst)"""

In [5]: print some_code
from baron import parse
from baron.helpers import show_node
fst = parse("a = 1")
show_node(fst)

In [6]: tree = parse(some_code)

In [7]: node = position_to_node(tree, (3, 8))

In [8]: show_node(node)
"parse"

In [9]: path = position_to_path(tree, (3, 8))

In [10]: path
Out[10]: [4, 'value', 'value', 0, 'value']
```

The first one gives the node and the second one the node's path in the tree. The latter tells you that to get to the node, you must take the 4th index of the root ListNode, followed twice by the "value" key of first the "assignment" Node and next the "atomtrailers" Node. Finally, take the 0th index in the resulting ListNode:

```
In [11]: show_node(tree[4]["value"]["value"][0])
{
    "type": "name",
    "value": "parse"
}
```

Neat. This is so common that there is a function to do that:

```
In [12]: from baron.path import path_to_node

In [13]: show_node(path_to_node(tree, path))
"parse"
```

With the two above, that's a total of three functions to locate a node.

You can also locate easily a "constant" node like a left parenthesis in a `funcdef` node:

```
In [14]: from baron.path import position_to_path

In [15]: fst = parse("a(1)")

In [16]: position_to_path(fst, (1, 1))
Out[16]: [0, 'value', 0, 'value']

In [17]: position_to_path(fst, (1, 2))
Out[17]: [0, 'value', 1, '(']

In [18]: position_to_path(fst, (1, 3))
Out[18]: [0, 'value', 1, 'value', 0, 'value', 'value']

In [19]: position_to_path(fst, (1, 4))
Out[19]: [0, 'value', 1, ')']
```

By the way, out of bound positions are handled gracefully:

```
In [20]: print(position_to_node(fst, (-1, 1)))
None

In [21]: print(position_to_node(fst, (1, 0)))
None

In [22]: print(position_to_node(fst, (1, 5)))
None

In [23]: print(position_to_node(fst, (2, 4)))
None
```

6.2.2 Bounding Box

Sometimes you want to know what are the left most and right most position of a rendered node or part of it. It is not a trivial task since you do not know easily each rendered line's length. That's why baron provides two helpers:

- `node_to_bounding_box(fst)`
- `path_to_bounding_box(fst, path)`

Examples are worth a thousand words so:

```
In [24]: from baron.path import node_to_bounding_box, path_to_bounding_box

In [25]: from baron import dumps

In [26]: fst = parse("a(1)\nb(2)")

In [27]: fst
Out[27]:
[{'type': 'atomtrailers',
  'value': [{'type': 'name', 'value': 'a'},
   {'first_formatting': [],
    'fourth_formatting': [],
    'second_formatting': [],
    'third_formatting': [],
    'type': 'call',
    'value': [{'first_formatting': [],
      'second_formatting': [],
      'target': {},
      'type': 'call_argument',
      'value': {'section': 'number', 'type': 'int', 'value': '1'}}]}]},
 {'formatting': [], 'indent': '', 'type': 'endl', 'value': '\n'},
 {'type': 'atomtrailers',
  'value': [{'type': 'name', 'value': 'b'},
   {'first_formatting': [],
    'fourth_formatting': [],
    'second_formatting': [],
    'third_formatting': [],
    'type': 'call',
    'value': [{'first_formatting': [],
      'second_formatting': [],
      'target': {},
      'type': 'call_argument',
      'value': {'section': 'number', 'type': 'int', 'value': '2'}}]}]}]

In [28]: print dumps(fst)
a(1)
b(2)

In [29]: node_to_bounding_box(fst)
Out[29]: BoundingBox (Position (1, 1), Position (2, 4))

In [30]: path_to_bounding_box(fst, [])
Out[30]: BoundingBox (Position (1, 1), Position (2, 4))

In [31]: fst[0]
Out[31]:
{'type': 'atomtrailers',
 'value': [{'type': 'name', 'value': 'a'},
  {'first_formatting': [],
   'fourth_formatting': [],
   'second_formatting': [],
   'third_formatting': [],
   'type': 'call',
```

```
   'value': [{'first_formatting': [],
     'second_formatting': [],
     'target': {},
     'type': 'call_argument',
     'value': {'section': 'number', 'type': 'int', 'value': '1'}}]}]}

In [32]: print dumps(fst[0])
a(1)

In [33]: node_to_bounding_box(fst[0])
Out[33]: BoundingBox (Position (1, 1), Position (1, 4))

In [34]: path_to_bounding_box(fst, [0])
Out[34]: BoundingBox (Position (1, 1), Position (1, 4))

In [35]: fst[0]["value"]
Out[35]:
[{'type': 'name', 'value': 'a'},
 {'first_formatting': [],
  'fourth_formatting': [],
  'second_formatting': [],
  'third_formatting': [],
  'type': 'call',
  'value': [{'first_formatting': [],
    'second_formatting': [],
    'target': {},
    'type': 'call_argument',
    'value': {'section': 'number', 'type': 'int', 'value': '1'}}]}]

In [36]: print dumps(fst[0]["value"])
a(1)

In [37]: node_to_bounding_box(fst[1])
Out[37]: BoundingBox (Position (1, 1), Position (2, 0))

In [38]: path_to_bounding_box(fst, [1])
Out[38]: BoundingBox (Position (1, 5), Position (2, 0))

In [39]: fst[0]["value"][1]
Out[39]:
{'first_formatting': [],
 'fourth_formatting': [],
 'second_formatting': [],
 'third_formatting': [],
 'type': 'call',
 'value': [{'first_formatting': [],
   'second_formatting': [],
   'target': {},
   'type': 'call_argument',
   'value': {'section': 'number', 'type': 'int', 'value': '1'}}]}

In [40]: print dumps(fst[0]["value"][1])
(1)

In [41]: node_to_bounding_box(fst[0]["value"][1])
Out[41]: BoundingBox (Position (1, 1), Position (1, 3))

In [42]: path_to_bounding_box(fst, [0, "value", 1])
```

```
Out[42]: BoundingBox (Position (1, 2), Position (1, 4))

In [43]: fst[0]["value"][1]["value"]
Out[43]:
[{'first_formatting': [],
  'second_formatting': [],
  'target': {},
  'type': 'call_argument',
  'value': {'section': 'number', 'type': 'int', 'value': '1'}}]

In [44]: print dumps(fst[0]["value"][1]["value"])
1

In [45]: node_to_bounding_box(fst[0]["value"][1]["value"])
Out[45]: BoundingBox (Position (1, 1), Position (1, 1))

In [46]: path_to_bounding_box(fst, [0, "value", 1, "value"])
Out[46]: BoundingBox (Position (1, 3), Position (1, 3))
```

The bounding box's *top_left* and *bottom_right* positions follow the same convention as for when locating a node: the line and column start at 1.

As you can see, the major difference between the two functions is that `node_to_bounding_box` will always give a left position of `(1, 1)` since it considers you want the bounding box of the whole node while `path_to_bounding_box` takes the location of the node in the fst into account.

6.3 Rendering the FST

This section is quite advanced and you will maybe never need to use what is in here. But if you want to process the whole rendered fst or part of it as a chunk, please read along since several helpers are provided.

6.3.1 Understanding core rendering

Baron renders the FST back into source code by following the instructions given by the `nodes_rendering_order` dictionary. It gives, for every FST node, the order in which the node components must be rendered and the nature of those components.

```
In [1]: from baron import nodes_rendering_order, parse

In [2]: from baron.helpers import show_node

In [3]: nodes_rendering_order["name"]
Out[3]: [('string', 'value', True)]

In [4]: show_node(parse("a_name")[0])
{
    "type": "name",
    "value": "a_name"
}

In [5]: nodes_rendering_order["tuple"]
Out[5]:
[('formatting', 'first_formatting', 'with_parenthesis'),
 ('constant', '(', 'with_parenthesis'),
```

```
 ('formatting', 'second_formatting', 'with_parenthesis'),
 ('list', 'value', True),
 ('formatting', 'third_formatting', 'with_parenthesis'),
 ('constant', ')', 'with_parenthesis'),
 ('formatting', 'fourth_formatting', 'with_parenthesis'),
 ('bool', 'with_parenthesis', False)]

In [6]: show_node(parse("(a_name,another_name,yet_another_name)")[0])
{
    "first_formatting": [],
    "fourth_formatting": [],
    "value": [
        {
            "type": "name",
            "value": "a_name"
        },
        {
            "first_formatting": [],
            "type": "comma",
            "second_formatting": []
        },
        {
            "type": "name",
            "value": "another_name"
        },
        {
            "first_formatting": [],
            "type": "comma",
            "second_formatting": []
        },
        {
            "type": "name",
            "value": "yet_another_name"
        }
    ],
    "second_formatting": [],
    "third_formatting": [],
    "with_parenthesis": true,
    "type": "tuple"
}

In [7]: nodes_rendering_order["comma"]
Out[7]:
[('formatting', 'first_formatting', True),
 ('constant', ',', True),
 ('formatting', 'second_formatting', True)]
```

For a "name" node, it is a list containing a unique component stored in a tuple but it can contain multiple ones like for a "tuple" node.

To render a node, you just need to render each element of the list, one by one, in the given order. As you can see, they are all formatted as a 3-tuple. The first column is the type which is one of the following:

```
In [8]: from baron.render import node_types

In [9]: node_types
Out[9]: {'bool', 'constant', 'formatting', 'key', 'list', 'node', 'string'}
```

With the exception of the "constant" node, the second column contains the key of the FST node which must be rendered. The first column explains how that key must be rendered. We'll see the third column later.

- A `node` node is one of the nodes in the `nodes_rendering_order` we just introduced, it is rendered by following the rules mentionned here. This is indeed a recursive definition.
- A `key` node is a branch of the tree that contains another node (a python dictionary).
- A `string` node is a leaf of the tree that contains a variable value, like the name of a function. former case, it is rendered by rendering its content.
- A `list` node is like a `key` node but can contain 0, 1 or several other nodes stored in a python list. For example, Baron root node is a `list` node since a python program is a list of statements. It is rendered by rendering each of its elements in order.
- A `formatting` node is similar in behaviour to a `list` node but contains only formatting nodes. This is basically where Baron distinguish itself from other ASTs.
- A `constant` node is a leaf of the FST tree. The second column always contain a string which is outputted directly. Compared to a `string` node, the `constant` node is identical for every instance of the nodes (e.g. the left parenthesis character `(` in a function call node or the `def` keyword of a function definition) while the `string` node's value can change (e.g. the name of the function in a function definition node).
- A `bool` node is a node used exclusively for conditional rendering. It's exact use will be explained later on with the tuple's third column but the main point for now is to know that they are never rendered.

Walktrough

Let's see all this is in action by rendering a "lambda" node. First, the root node is always a "list" node and since we are only parsing one statement, the root node contains our "lambda" node at index 0:

```
In [10]: fst = parse("lambda x, y = 1: x + y")

In [11]: fst[0]["type"]
Out[11]: 'lambda'
```

For reference, you can find the (long) FST produced by the lambda node at the end of this section.

Now, let's see how to render a "lambda" node:

```
In [12]: nodes_rendering_order["lambda"]
Out[12]:
[('constant', 'lambda', True),
 ('formatting', 'first_formatting', True),
 ('list', 'arguments', True),
 ('formatting', 'second_formatting', True),
 ('constant', ':', True),
 ('formatting', 'third_formatting', True),
 ('key', 'value', True)]
```

Okay, first the string constant "lambda", then a first_formatting node which represents the space between the string "lambda" and the variable "x".

```
In [13]: fst[0]["first_formatting"]
Out[13]: [{'type': 'space', 'value': ' '}]
```

The "first_formatting" contains a list whose unique element is a "space" node.

```
In [14]: fst[0]["first_formatting"][0]
Out[14]: {'type': 'space', 'value': ' '}

In [15]: nodes_rendering_order["space"]
Out[15]: [('string', 'value', True)]
```

Which in turn is rendered by printing the value of the string of the space node.

```
In [16]: fst[0]["first_formatting"][0]["value"]
Out[16]: ' '
```

So far we have outputted "lambda ". Tedious but exhaustive.

We have exhausted the "first_formatting" node so we go back up the tree. Next is the "list" node representing the arguments:

```
In [17]: fst[0]["arguments"]
Out[17]:
[{'first_formatting': [],
  'second_formatting': [],
  'target': {'type': 'name', 'value': 'x'},
  'type': 'def_argument',
  'value': {}},
 {'first_formatting': [],
  'second_formatting': [{'type': 'space', 'value': ' '}],
  'type': 'comma'},
 {'first_formatting': [{'type': 'space', 'value': ' '}],
  'second_formatting': [{'type': 'space', 'value': ' '}],
  'target': {'type': 'name', 'value': 'y'},
  'type': 'def_argument',
  'value': {'section': 'number', 'type': 'int', 'value': '1'}}]
```

Rendering a "list" node is done one element at a time. First a "def_argument", then a "comma" and again a "def_argument".

```
In [18]: fst[0]["arguments"][0]
Out[18]:
{'first_formatting': [],
 'second_formatting': [],
 'target': {'type': 'name', 'value': 'x'},
 'type': 'def_argument',
 'value': {}}

In [19]: nodes_rendering_order["def_argument"]
Out[19]:
[('key', 'target', True),
 ('formatting', 'first_formatting', 'value'),
 ('constant', '=', 'value'),
 ('formatting', 'second_formatting', 'value'),
 ('key', 'value', 'value')]
```

The first "def_argument" is rendered by first outputting the content of a name "string" node:

```
In [20]: fst[0]["arguments"][0]["name"]

KeyErrorTraceback (most recent call last)
<ipython-input-20-7e5f373e673d> in <module>()
----> 1 fst[0]["arguments"][0]["name"]
```

```
KeyError: 'name'
```

Now, we have outputted "lambda x". At first glance we could say we should render the second element of the "def_argument" node but as we'll see in the next section, it is not the case because of the third column of the tuple.

For reference, the FST of the lambda node:

```
In [21]: show_node(fst[0])
{
    "first_formatting": [
        {
            "type": "space",
            "value": " "
        }
    ],
    "value": {
        "first_formatting": [
            {
                "type": "space",
                "value": " "
            }
        ],
        "value": "+",
        "second_formatting": [
            {
                "type": "space",
                "value": " "
            }
        ],
        "second": {
            "type": "name",
            "value": "y"
        },
        "type": "binary_operator",
        "first": {
            "type": "name",
            "value": "x"
        }
    },
    "second_formatting": [],
    "third_formatting": [
        {
            "type": "space",
            "value": " "
        }
    ],
    "arguments": [
        {
            "first_formatting": [],
            "type": "def_argument",
            "target": {
                "type": "name",
                "value": "x"
            },
            "value": {},
            "second_formatting": []
        },
```

```
        {
            "first_formatting": [],
            "type": "comma",
            "second_formatting": [
                {
                    "type": "space",
                    "value": " "
                }
            ]
        },
        {
            "first_formatting": [
                {
                    "type": "space",
                    "value": " "
                }
            ],
            "type": "def_argument",
            "target": {
                "type": "name",
                "value": "y"
            },
            "value": {
                "section": "number",
                "type": "int",
                "value": "1"
            },
            "second_formatting": [
                {
                    "type": "space",
                    "value": " "
                }
            ]
        }
    ],
    "type": "lambda"
}
```

Dependent rendering

Sometimes, some node elements must not be outputted. In our "def_argument" example, all but the first are conditional. They are only rendered if the FST's "value" node exists and is not empty. Let's compare the two "def_arguments" FST nodes:

```
In [22]: fst[0]["arguments"][0]
Out[22]:
{'first_formatting': [],
 'second_formatting': [],
 'target': {'type': 'name', 'value': 'x'},
 'type': 'def_argument',
 'value': {}}

In [23]: fst[0]["arguments"][2]
Out[23]:
{'first_formatting': [{'type': 'space', 'value': ' '}],
 'second_formatting': [{'type': 'space', 'value': ' '}],
```

```
 'target': {'type': 'name', 'value': 'y'},
 'type': 'def_argument',
 'value': {'section': 'number', 'type': 'int', 'value': '1'}}

In [24]: nodes_rendering_order[fst[0]["arguments"][2]["type"]]
Out[24]:
[('key', 'target', True),
 ('formatting', 'first_formatting', 'value'),
 ('constant', '=', 'value'),
 ('formatting', 'second_formatting', 'value'),
 ('key', 'value', 'value')]
```

The "value" is empty for the former "def_argument" but not for the latter because it has a default value of "= 1".

```
In [25]: from baron import dumps

In [26]: dumps(fst[0]["arguments"][0])
Out[26]: 'x'

In [27]: dumps(fst[0]["arguments"][2])
Out[27]: 'y = 1'
```

The rule here is that the third column of a node is one of:

- True, it is always rendered;
- False, it is never rendered;
- A string, it is rendered conditionnally. It is not rendered if the key it references is either empty or False. It also must reference an existing key. In our example above, it references the existing "value" key which is empty in the first case and not empty in the second.

This is how "bool" nodes are never outputted: their third column is always False.

We will conclude here now that we have seen an example of every aspect of FST rendering. Understanding everything is not required to use Baron since several helpers like `render`, `RenderWalker` or `dumps` handle all the complexity under the hood.

6.3.2 Render Helper

Baron provides a render function helper which walks recursively the `nodes_rendering_order` dictionnary for you:

`baron.render.`**`render`**(*node*, *strict=False*)

Recipe to render a given FST node.

The FST is composed of branch nodes which are either lists or dicts and of leaf nodes which are strings. Branch nodes can have other list, dict or leaf nodes as childs.

To render a string, simply output it. To render a list, render each of its elements in order. To render a dict, you must follow the node's entry in the nodes_rendering_order dictionary and its dependents constraints.

This function hides all this algorithmic complexity by returning a structured rendering recipe, whatever the type of node. But even better, you should subclass the RenderWalker which simplifies drastically working with the rendered FST.

The recipe is a list of steps, each step correspond to a child and is actually a 3-uple composed of the following fields:

- *key_type* is a string determining the type of the child in the second field (*item*) of the tuple. It can be one of:
  - 'constant': the child is a string
  - 'node': the child is a dict
  - 'key': the child is an element of a dict
  - 'list': the child is a list
  - 'formatting': the child is a list specialized in formatting
- *item* is the child itself: either a string, a dict or a list.
- *render_key* gives the key used to access this child from the parent node. It's a string if the node is a dict or a number if its a list.

Please note that "bool" *key_types* are never rendered, that's why they are not shown here.

6.3.3 RenderWalker Helper

But even easier, Baron provides a walker class whose job is to walk the fst while rendering it and to call user-provided callbacks at each step:

class baron.render.**RenderWalker**(*strict=False*)

Inherit me and overload the methods you want.

When calling walk() on a FST node, this class will traverse all the node's subtree by following the recipe given by the *render* function for the node and recursively for all its childs. At each recipe step, it will call methods that you can override to make a specific process.

For every "node", "key", "list", "formatting" and "constant" childs, it will call the *before* method when going down the tree and the *after* method when going up. There are also specific *before_[node,key,list,formatting,constant]* and *after_[node,key,list,formatting,constant]* methods provided for convenience.

The latter are called on specific steps:

- before_list: called before encountering a list of nodes
- after_list: called after encountering a list of nodes
- before_formatting: called before encountering a formatting list
- after_formatting: called after encountering a formatting list
- before_node: called before encountering a node
- after_node: called after encountering a node
- before_key: called before encountering a key type entry
- after_key: called after encountering a key type entry
- before_leaf: called before encountering a leaf of the FST (can be a constant (like "def" in a function definition) or an actual value like the value a name node)
- after_leaf: called after encountering a leaf of the FST (can be a constant (like "def" in a function definition) or an actual value like the value a name node)

Every method has the same signature: (self, node, render_pos, render_key).

Internally, Baron uses the `RenderWalker` for multiple tasks like for the `dumps` function:

```
from baron.render import RenderWalker

def dumps(tree):
    return Dumper().dump(tree)

class Dumper(RenderWalker):
    def before_constant(self, constant, key):
        self.dump += constant

    def before_string(self, string, key):
        self.dump += string

    def dump(self, tree):
        self.dump = ''
        self.walk(tree)
        return self.dump
```

As you can see it is quite simple since it only needs the `before_constant` and the `before_string` methods with the same exact code.

6.3.4 PathWalker Helper

If while walking you need to know the current path of the node, then you should subclass `PathWalker` instead:

class baron.path.**PathWalker**(*strict=False*)

Gives the current path while walking the rendered tree

It adds an attribute "current_path" which is updated each time the walker takes a step.

Here is a succint example of what you should expect when using the `PathWalker`:

```
In [28]: from baron.path import PathWalker

In [29]: fst = parse("a = 1")

In [30]: class PathWalkerPrinter(PathWalker):
   ....:     def before(self, key_type, item, render_key):
   ....:         super(PathWalkerPrinter, self).before(key_type, item, render_key)
   ....:         print(self.current_path)
   ....:

In [31]: def after(self, key_type, item, render_key):
   ....:         print(self.current_path)
   ....:         super(PathWalkerPrinter, self).after(key_type, item, render_key)
   ....:
```

Like in the example, don't forget to call the before and after methods of the parent class. Furthermore, you need to respect the order specified above, that is:

- Calling `super().before()` should be done before your code using the `self.path` attribute.
- Calling `super().after()` should be done after your code using the `self.path` attribute.

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

Index

P

PathWalker (class in baron.path), 26

R

render() (in module baron.render), 24
RenderWalker (class in baron.render), 25