Baron Documentation

Release 0.2

Laurent Peuch

June 10, 2014

Contents

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`.

CHAPTER 1

Installation

```
pip install baron
```

CHAPTER 2

Github (source, bug tracker, etc.)

https://github.com/psycojoker/baron

CHAPTER 3

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.

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 = "a = 1"

In [3]: fst = parse(source_code)

In [4]: generated_source_code = dumps(fst)

In [5]: generated_source_code
Out[5]: 'a = 1'

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'
```

4.1 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.

4.1.1 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": null,
        "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"
    }
]
```

4.1.2 Show_file

`show_file` is used on a file path:

```
from baron.helpers import show_file

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

4.1.3 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": null,
        "type": "assignment"
    }
]
```

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

4.2 Rendering the FST

Baron renders the FST back into source code by following the instructions given by the `nodes_rendering_order` dictionary. It gives, for everynode FST node, the order in which the node must be rendered.

```
In [16]: from baron import nodes_rendering_order

In [17]: from baron.helpers import show_node

In [18]: nodes_rendering_order["name"]
Out[18]: [('key', 'value', True)]

In [19]: show_node(parse("a_name")[0])
{
    "type": "name",
    "value": "a_name"
}

In [20]: nodes_rendering_order["tuple"]
Out[20]:
[('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')]

In [21]: 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 [22]: nodes_rendering_order["comma"]
Out[22]:
[('formatting', 'first_formatting', True),
 ('constant', ',', True),
 ('formatting', 'second_formatting', True)]
```

For a "name" node, it is a list containing an unique 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 [23]: from baron.render import node_types

In [24]: node_types
Out[24]: {'constant', 'formatting', 'key', 'list', 'node'}
```

Apart for 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 either a branch of the tree if the corresponding FST node's key contains another node or a leaf if it contains a string. In the former case, it is rendered by rendering its content. In the latter, the string is outputted directly.
- A `list` node is like a `key` node but can contain 0, 1 or several other nodes. 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 ASTs.
- A `constant` node is a leaf of the FST tree. The second column always contains a string which is outputted directly. Compared to a `key` node containing a string, the `constant` node is identical for every instance of the nodes (e.g. the left parenthesis character `(` in a function call node of the `def` keyword of a function definition) while the `key` node's value can change (e.g. the name of the function in a function call node).

4.2.1 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 [25]: fst = parse("lambda x, y = 1: x + y")

In [26]: fst[0]["type"]
Out[26]: '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 [27]: nodes_rendering_order["lambda"]
Out[27]:
[('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 [28]: fst[0]["first_formatting"]
Out[28]: [{'type': 'space', 'value': ' '}]
```

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

```
In [29]: fst[0]["first_formatting"][0]
Out[29]: {'type': 'space', 'value': ' '}

In [30]: nodes_rendering_order["space"]
Out[30]: [('key', 'value', True)]
```

Which in turn is rendered by looking at the value key of the space node. It's a string so it is outputted directly.

```
In [31]: fst[0]["first_formatting"][0]["value"]
Out[31]: ' '
```

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 [32]: fst[0]["arguments"]
Out[32]:
[{'first_formatting': [],
  'name': 'x',
  'second_formatting': [],
  'type': 'def_argument',
  'value': {}},
 {'first_formatting': [],
  'second_formatting': [{'type': 'space', 'value': ' '}],
  'type': 'comma'},
 {'first_formatting': [{'type': 'space', 'value': ' '}],
  'name': 'y',
  'second_formatting': [{'type': 'space', 'value': ' '}],
  '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 [33]: fst[0]["arguments"][0]
Out[33]:
{'first_formatting': [],
 'name': 'x',
 'second_formatting': [],
 'type': 'def_argument',
 'value': {}}

In [34]: nodes_rendering_order["def_argument"]
Out[34]:
```

```
[('key', 'name', 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 "key" node, which is string and thus outputted directly:

```
In [35]: fst[0]["arguments"][0]["name"]
Out[35]: 'x'
```

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 thanks to the third column of the tuple.

For reference, the FST of the lambda node:

```
In [36]: 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",
            "name": "x",
            "value": {},
            "second_formatting": []
        },
        {
            "first_formatting": [],
            "type": "comma",
            "second_formatting": [
                {
                    "type": "space",
                    "value": " "
                }
            ]
        },
        {
            "first_formatting": [
                {
                    "type": "space",
                    "value": " "
                }
            ],
            "type": "def_argument",
            "name": "y",
            "value": {
                "section": "number",
                "type": "int",
                "value": "1"
            },
            "second_formatting": [
                {
                    "type": "space",
                    "value": " "
                }
            ]
        }
    ],
    "type": "lambda"
}
```

4.2.2 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 [37]: fst[0]["arguments"][0]
Out[37]:
{'first_formatting': [],
 'name': 'x',
 'second_formatting': [],
 'type': 'def_argument',
 'value': {}}

In [38]: fst[0]["arguments"][2]
Out[38]:
{'first_formatting': [{'type': 'space', 'value': ' '}],
 'name': 'y',
 'second_formatting': [{'type': 'space', 'value': ' '}],
 'type': 'def_argument',
 'value': {'section': 'number', 'type': 'int', 'value': '1'}}

In [39]: nodes_rendering_order[fst[0]["arguments"][2]["type"]]
Out[39]:
[('key', 'name', 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 only the latter has a default assignment "= 1".

```
In [40]: dumps(fst[0]["arguments"][0])
Out[40]: 'x'

In [41]: dumps(fst[0]["arguments"][2])
Out[41]: 'y = 1'
```

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 `dumps` handles all the complexity.

4.3 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` and `position_to_path`. Both functions 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 dictionary in JSON format which contains 3 values:

- the `path` key contains the path: a list of int and strings which represent either the key to take in a Node or the index in a ListNode (e.g. "target", "value", 0)
- the `type` key tells the type of the FST node (e.g. "function", "assignment", "class")
- the `position_in_rendering_list` key is the rendering position of the node compared to its parent node. This is especially needed when the character pointed on is actually not a node itself but only a part of a parent node. It's a little complicated but don't worry, examples will follow.

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

```
In [42]: from baron import parse

In [43]: from baron.finder import position_to_node, position_to_path
---------------------------------------------------------------------------
ImportError                               Traceback (most recent call last)
<ipython-input-43-2ee5cf0b0d42> in <module>()
----> 1 from baron.finder import position_to_node, position_to_path
```

```
ImportError: No module named finder

In [44]: some_code = """\
   ....: from baron import parse
   ....: from baron.helpers import show_node
   ....: fst = parse("a = 1")
   ....: show_node(fst)
   ....: """
   ....:

In [45]: tree = parse(some_code)

In [46]: node = position_to_node(tree, 3, 8)
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-46-6fcd4b70f12a> in <module>()
----> 1 node = position_to_node(tree, 3, 8)

NameError: name 'position_to_node' is not defined

In [47]: show_node(node)
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-47-478762d152dd> in <module>()
----> 1 show_node(node)

NameError: name 'node' is not defined

In [48]: path = position_to_path(tree, 3, 8)
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-48-45ebd8ca0401> in <module>()
----> 1 path = position_to_path(tree, 3, 8)

NameError: name 'position_to_path' is not defined

In [49]: path
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-49-e7fa32cb05ba> in <module>()
----> 1 path

NameError: name 'path' is not defined
```

The first one gives the node and the second one the node path. Both also give its type but what does the keys in the path correspond to exactly? The path 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 [50]: 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 [51]: from baron.finder import path_to_node
---------------------------------------------------------------------------
ImportError                               Traceback (most recent call last)
<ipython-input-51-7499ebbbb11a> in <module>()
----> 1 from baron.finder import path_to_node

ImportError: No module named finder

In [52]: show_node(path_to_node(tree, path))
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-52-d1f4f0a57623> in <module>()
----> 1 show_node(path_to_node(tree, path))

NameError: name 'path_to_node' is not defined
```

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

And what about the `position_in_rendering_list`? To understand, the best is an example. What happens if you try to locate the node corresponding to the left parenthesis on line 3?

```
In [53]: position_to_path(tree, 3, 12)
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-53-d1124224f778> in <module>()
----> 1 position_to_path(tree, 3, 12)

NameError: name 'position_to_path' is not defined

In [54]: show_node(tree[4]["value"]["value"][1])
{
    "first_formatting": [],
    "fourth_formatting": [],
    "value": [
        {
            "first_formatting": [],
            "type": "call_argument",
            "name": {},
            "value": {
                "first_formatting": [],
                "type": "string",
                "value": "\"a = 1\"",
                "second_formatting": []
            },
            "second_formatting": []
        }
    ],
    "second_formatting": [],
    "third_formatting": [],
    "type": "call"
}
```

As you can see, the information given by the path is that I'm on a call node. No parenthesis in sight. That's where the `position_in_rendering_list` proves useful. It tells you where you are located in the rendering dictionary:

```
In [55]: from baron import nodes_rendering_order

In [56]: nodes_rendering_order["call"]
Out[56]:
```

```
[('formatting', 'first_formatting', True),
 ('constant', '(', True),
 ('formatting', 'second_formatting', True),
 ('key', 'value', True),
 ('formatting', 'third_formatting', True),
 ('constant', ')', True),
 ('formatting', 'fourth_formatting', True)]

In [57]: nodes_rendering_order["call"][1]
Out[57]: ('constant', '(', True)
```

Because the parenthesis is a constant, there is no specific node for the parenthesis. So the path can only go as far as the parent node, here "call", and show you the position in the rendering dictionary.

For example, it allows you to distinguish the left and right parenthesis in a call.

```
In [58]: position_to_path(tree, 3, 20)
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-58-4aae88753679> in <module>()
----> 1 position_to_path(tree, 3, 20)

NameError: name 'position_to_path' is not defined

In [59]: nodes_rendering_order["call"][5]
Out[59]: ('constant', ')', True)
```

To conclude this section, let's look at a last example of path:

```
In [60]: from baron.finder import position_to_path
---------------------------------------------------------------------------
ImportError                               Traceback (most recent call last)
<ipython-input-60-e5e6945dbeee> in <module>()
----> 1 from baron.finder import position_to_path

ImportError: No module named finder

In [61]: fst = parse("a(1)")

In [62]: position_to_path(fst, 1, 1)
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-62-a63380ebd3d9> in <module>()
----> 1 position_to_path(fst, 1, 1)

NameError: name 'position_to_path' is not defined

In [63]: position_to_path(fst, 1, 2)
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-63-dac281e2e154> in <module>()
----> 1 position_to_path(fst, 1, 2)

NameError: name 'position_to_path' is not defined

In [64]: position_to_path(fst, 1, 3)
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-64-89318ee1c241> in <module>()
```

```
----> 1 position_to_path(fst, 1, 3)

NameError: name 'position_to_path' is not defined

In [65]: position_to_path(fst, 1, 4)
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-65-1802b6b7fbd2> in <module>()
----> 1 position_to_path(fst, 1, 4)

NameError: name 'position_to_path' is not defined
```

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

```
In [66]: print(position_to_node(fst, -1, 1))
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-66-4de3017656f3> in <module>()
----> 1 print(position_to_node(fst, -1, 1))

NameError: name 'position_to_node' is not defined

In [67]: print(position_to_node(fst, 1, 0))
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-67-853480996a4b> in <module>()
----> 1 print(position_to_node(fst, 1, 0))

NameError: name 'position_to_node' is not defined

In [68]: print(position_to_node(fst, 1, 5))
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-68-b11a65655afb> in <module>()
----> 1 print(position_to_node(fst, 1, 5))

NameError: name 'position_to_node' is not defined

In [69]: print(position_to_node(fst, 2, 4))
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-69-df6dcdea554f> in <module>()
----> 1 print(position_to_node(fst, 2, 4))

NameError: name 'position_to_node' is not defined
```

CHAPTER 5

RenderWalker

Internally, Baron uses a walker to traverse a FST tree, it's a generic class that you are free to use. To do so, you inherit from it and overload the chosen methods. You then launch an instance using it's `walk` method. Here is how the `Dumper` (called by the function `dumps`) is written using it:

```
In [70]: from baron.render import RenderWalker

In [71]: class Dumper(RenderWalker):
   ....:     """Usage: Dumper().dump(tree)"""
   ....:     def on_leaf(self, constant, pos, key):
   ....:         self.dump += constant
   ....:         return self.CONTINUE
   ....:     def dump(self, tree):
   ....:         self.dump = ''
   ....:         self.walk(tree)
   ....:         return self.dump
   ....:
```

The available methods that you can overload are:

- `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
- `on_leaf` called when 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)`.