RedBaron Documentation

Release 0.9.2

Laurent Peuch

Aug 11, 2019

Contents

1 Introduction **1**

2 A note about the examples **3**

3 Code **5**

4 Installation **7**

5 Basic usage **9**

6 Financial support **11**
- 6.1 Our supporters . . . 11

7 Table of content **13**
- 7.1 Learn how to use RedBaron . . . 13
- 7.2 Why is this important? . . . 25
- 7.3 Basics . . . 26
- 7.4 Querying . . . 32
- 7.5 Modifying . . . 38
- 7.6 Proxy List . . . 41
- 7.7 Other . . . 51

8 Reference **69**
- 8.1 Nodes References Page . . . 69
- 8.2 TopClass . . . 69
- 8.3 Nodes . . . 71

9 Indices and tables **127**

Index **129**

CHAPTER 1

Introduction

RedBaron is a python library and tool powerful enough to be used into IPython solely that intent to make the process of **writing code that modify source code** as easy and as simple as possible. That include writing custom refactoring, generic refactoring, tools, IDE or directly modifying you source code into IPython with a higher and more powerful abstraction than the advanced texts modification tools that you find in advanced text editors and IDE.

RedBaron guaranteed you that **it will only modify your code where you ask him to**. To achieve this, it is based on Baron a lossless AST for Python that guarantees the operation `fst_to_code(code_to_fst(source_code)) == source_code`.

RedBaron API and feel is heavily inspired by BeautifulSoup. It tries to be simple and intuitive and that once you've get the basics principles, you are good without reading the doc for 80% of your operations.

CHAPTER 2

A note about the examples

This documentation is full of example for nearly everything. But in fact, those aren't really "example": those are real life code that are executed at the compilation time of this documentation, this guaranteed the example you see to work exactly the same way for you.

Funny side effect: this make it possible to "break" this documentation.

CHAPTER 3

Code

https://github.com/PyCQA/redbaron

CHAPTER 4

Installation

```
pip install redbaron[pygments]
```

Or if you don't want to have syntax highlight in your shell or don't need it:

```
pip install redbaron
```

CHAPTER 5

Basic usage

Simple API: give string, get string back.

```
In [1]: from redbaron import RedBaron

In [2]: red = RedBaron("some_value = 42")

In [3]: red.dumps()  # get code back
Out[3]: 'some_value = 42'
```

Though to be used in IPython directly:

```
In [4]: red  # direct feedback like BeautifulSoup, "0" here is the index of the node␣
↪in our source code
Out[4]: 0   some_value = 42

In [5]: red.help()  # helper function that describe nodes content so you don't have␣
↪to read the doc
0 -----------------------------------------------------
AssignmentNode()
  # identifiers: assign, assignment, assignment_, assignmentnode
  operator=''
  target ->
    NameNode()
      # identifiers: name, name_, namenode
      value='some_value'
  annotation ->
    None
  value ->
    IntNode()
      # identifiers: int, int_, intnode
      value='42'
```

Easy nodes modifications, you already know how to code in python, so pass python code (in a string) on the attribute you want to modify (wonder what `.value` is? look at the output of `.help()` in the previous example):

```
In [6]: red[0].value = "1 + 4"

In [7]: red
Out[7]: 0   some_value = 1 + 4
```

Easy queries, just like in BeautifulSoup:

```
In [8]: red.find("int", value=4)

In [9]: red.find_all("int")  # can also be written red("int") like in BeautifulSoup
Out[9]:
0   1
1   4
```

Queries can be very powerful, you can test each attributes with value/lambda/regex/special syntax for regex/globs.

Now let's pretend that we are editing a django settings.py (notice that we are extending our source code using the same API than the one of a python list since we are in a list of lines):

```
In [10]: red.extend(["\n", "INSTALLED_APPLICATIONS = (\n    'django',\n)"])  # here
↪"\n" is to had a blank line

In [11]: red
Out[11]:
0   some_value = 1 + 4
1   INSTALLED_APPLICATIONS = (
        'django',
    )
```

And let's install another django application! (again: same API than a python list)

```
In [12]: red.find("assignment", target=lambda x: x.dumps() == "INSTALLED_APPLICATIONS
↪").value.append("'another_app'")

In [13]: red
Out[13]:
0   some_value = 1 + 4
1   INSTALLED_APPLICATIONS = (
        'django',
        'another_app',
    )
```

Notice that the formatting of the tuple has been detected and respected when adding the new django application.

And let's see the result of our work:

```
In [14]: print(red.dumps())
some_value = 1 + 4
INSTALLED_APPLICATIONS = (
    'django',
    'another_app',
)
```

CHAPTER 6

Financial support

Baron and RedBaron are a very advanced piece of engineering that requires a lot of time of concentration to work on. Until the end of 2018, the development has been a full volunteer work mostly done by [Bram](https://github.com/psycojoker), but now, to reach the next level and bring those projects to the stability and quality you expect, we need your support.

You can join our contributors and sponsors on our transparent [OpenCollective](https://opencollective.com/redbaron), every contribution will count and will be mainly used to work on the projects stability and quality but also on continuing, on the side, the R&D side of those projects.

6.1 Our supporters

Become our first sponsor!

CHAPTER 7

Table of content

7.1 Learn how to use RedBaron

This tutorial guides you through the big principles of RedBaron and highlights the most useful helpers and tricks. It is more or less a lighter version of the already existing documentation.

A reminder before starting:

- **RedBaron doesn't do static analysis** and will never do (but it's very likely that it will be combined with tools that do it, like astroid or rope, to bring static analysis into RedBaron or easy source code modification in the others)

The structure of this tutorial is similar to the documentation's:

- basic principles and how to use it in a shell
- how to query the tree
- how to modify the tree
- how to play with list of things
- miscellaneous but useful stuff

7.1.1 Basic principles

Input and output with the source code in a string:

```
from redbaron import RedBaron

red = RedBaron("code source as a string")
red.dumps()
```

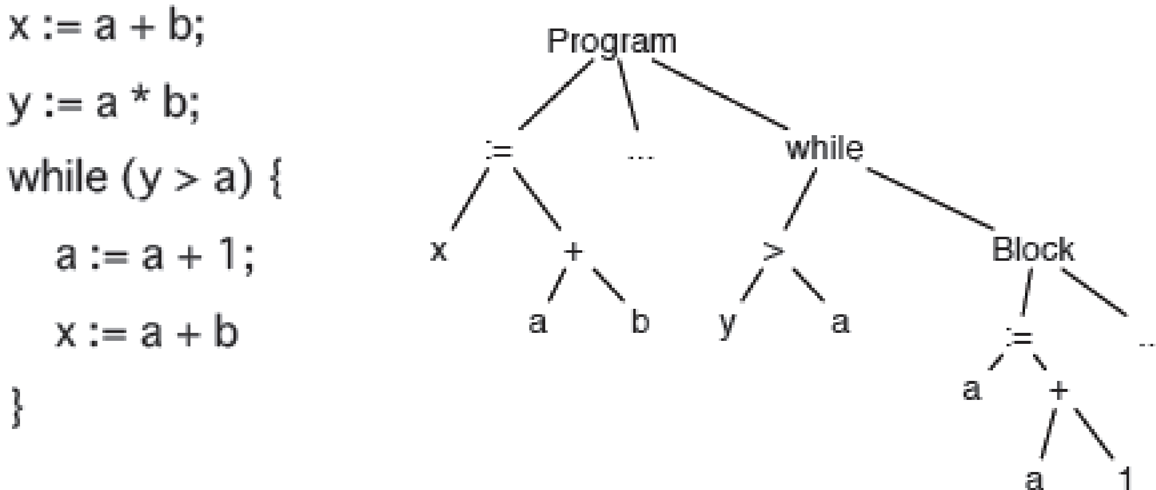
Input and output with the source code in a file:

```
from redbaron import RedBaron

with open("code.py", "r") as source_code:
    red = RedBaron(source_code.read())

with open("code.py", "w") as source_code:
    source_code.write(red.dumps())
```

Now that you know how to load your code into RedBaron, let's talk about its principles:

- RedBaron represents the source code as a tree. This is because when you are writing source code (of any classical language), you are actually writing a tree structure in the source file.
- For example: in `1 + 2` the top node is `+`, the left one is `1` and the right one is `2`.
- In `(1 + 2) + 3` the top node is, again, `+`, but the left one is actually `(1 + 2)` which is again another `+` node! This structure *is* a tree.
- The classical approach to handle such a structure is to use an Abstract Syntax Tree (AST) (it is used by compilers and interpreters like cpython).
- RedBaron, by relying on Baron, uses a *Full* Syntax Tree (FST). It's like an AST except it keeps every information, included formatting, and is then a lossless representation of the source code. Under the hood, the FST produced by Baron is in JSON and has been thought to be read and used by humans (although not as easily as RedBaron).
- So, when BeautifulSoup wraps the HTML datastructure into objects, RedBaron does the same thing for the FST datastructure and provides a nice way to interact with the source code.

Example of an AST for some language that looks like Go:



While you don't have to do that to use RedBaron on a daily basis, seeing the produced FST can help your understand RedBaron better (every key that has "_formatting" in its name is formatting related):

```
In [1]: import json

In [2]: red = RedBaron("1+2")

In [3]: print(json.dumps(red.fst(), indent=4))  # json.dumps is used for pretty␣
↪printing
[
```

(continues on next page)

(continued from previous page)

```
    {
        "first_formatting": [],
        "value": "+",
        "second_formatting": [],
        "second": {
            "section": "number",
            "type": "int",
            "value": "2"
        },
        "type": "binary_operator",
        "first": {
            "section": "number",
            "type": "int",
            "value": "1"
        }
    }
]
```

7.1.2 Use it in a shell

Now that we stated the concept of the source code as a tree, let's explore it.

First, like BeautifulSoup, when used in a shell RedBaron displays the currently selected source code, so you'll have a direct idea of what you are working on:

```
In [4]: red = RedBaron("stuff = 1 + 2\nprint 'Beautiful result:', stuff ")

In [5]: red
Out[5]:
0   stuff = 1 + 2
1   print 'Beautiful result:', stuff
2   ' '
```

You might notice the `0` and the `1` on the left: those are the indexes of the 2 nodes in the root of the source code. In fact, a source code is a list of statements so the root node `red` is a list. See by yourself:

```
In [6]: red[0]
Out[6]: stuff = 1 + 2

In [7]: red[1]
Out[7]: print 'Beautiful result:', stuff
```

But now, how to access the attributes? Since reading the doc for every node is boring, RedBaron comes with a helper method that shows you the underlying structure of the currently selected nodes:

```
In [8]: red[0]
Out[8]: stuff = 1 + 2

In [9]: red[0].help()
AssignmentNode()
  # identifiers: assign, assignment, assignment_, assignmentnode
  operator=''
  target ->
    NameNode()
      # identifiers: name, name_, namenode
```

(continues on next page)

(continued from previous page)

```
      value='stuff'
  annotation ->
    None
  value ->
    BinaryOperatorNode()
      # identifiers: binary_operator, binary_operator_, binaryoperator,␣
↪binaryoperatornode
      value='+'
      first ->
        IntNode() ...
      second ->
        IntNode() ...
```

The output might be a bit scary at first, but it's simply showing you the underlying structure, mapped to Baron JSON's one.

By the way, RedBaron has nice coloration features if you use ipython as your python shell.

Let's take it step by step:

- We are on an `AssignmentNode` (something like `a = b`) that has 3 attributes: `operator`, `target` and `value`.
- The `operator` is an empty string (it could have been a python operator like + in a case like `a += b`)
- `target` points to another node, a `NameNode` (you can see this thanks to the arrow `->` instead of an equal sign `=`)
- `value` points to a `BinaryOperatorNode`.

To get more information about all the existing kind of nodes, see the documentation: *Nodes References Page*.

Let's try it:

```
In [10]: red[0]
Out[10]: stuff = 1 + 2

In [11]: red[0].operator
Out[11]: ''

In [12]: red[0].target
Out[12]: stuff

In [13]: red[0].value
Out[13]: 1 + 2
```

For now we saw attributes that are either strings or pointing to other nodes, respectively called leafs and branches in the tree terminology. The last kind of attributes that you will encounter are a special case of the branch nodes: instead of pointing to a single node, they point to a list of nodes. You can see this in the print statement's `value` attribute:

```
In [14]: red[1].help()
PrintNode()
  # identifiers: print, print_, printnode
  destination ->
    None
  value ->
    * StringNode()
        # identifiers: string, string_, stringnode
        value="'Beautiful result:'"
```

(continues on next page)

(continued from previous page)

```
    * NameNode()
        # identifiers: name, name_, namenode
        value='stuff'
```

Notice the * before StringNode and NameNode? It indicates that they are items of a list. Look:

```
In [15]: red[1]
Out[15]: print 'Beautiful result:', stuff

In [16]: red[1].value
Out[16]:
0   'Beautiful result:'
1   stuff

In [17]: red[1].value[0]
Out[17]: 'Beautiful result:'

In [18]: red[1].value[1]
Out[18]: stuff
```

And if we show the help of the value attribute, we clearly see that there is a list of nodes.

```
In [19]: red[1].value.help()
0 -----------------------------------------------------
StringNode()
  # identifiers: string, string_, stringnode
  value="'Beautiful result:'"
1 -----------------------------------------------------
CommaNode()
  # identifiers: comma, comma_, commanode
2 -----------------------------------------------------
NameNode()
  # identifiers: name, name_, namenode
  value='stuff'
```

This is similar for the root node, which is itself also a list of nodes:

```
In [20]: red.help()
0 -----------------------------------------------------
AssignmentNode()
  # identifiers: assign, assignment, assignment_, assignmentnode
  operator=''
  target ->
    NameNode()
      # identifiers: name, name_, namenode
      value='stuff'
  annotation ->
    None
  value ->
    BinaryOperatorNode()
      # identifiers: binary_operator, binary_operator_, binaryoperator,␣
↪binaryoperatornode
      value='+'
      first ->
        IntNode() ...
      second ->
        IntNode() ...
```

(continues on next page)

(continued from previous page)

```
1 -----------------------------------------------------
EndlNode()
  # identifiers: endl, endl_, endlnode
  value='\n'
  indent=''
2 -----------------------------------------------------
PrintNode()
  # identifiers: print, print_, printnode
  destination ->
    None
  value ->
    * StringNode()
        # identifiers: string, string_, stringnode
        value="'Beautiful result:'"
    * NameNode()
        # identifiers: name, name_, namenode
        value='stuff'
3 -----------------------------------------------------
SpaceNode()
  # identifiers: space, space_, spacenode
  value=' '
```

And *voilà*, you now know how to navigate the tree by attributes without having to read any documentation!

If you're curious about the `identifiers` outputted by the `.help()` method, read on to the next section.

And one last thing: by default `.help()` stops at a certain "deepness level" and displays `...` instead of going further. To avoid that, simply pass an integer that indicates the "deepness level" you want, or give `True` if you want to display the whole tree.

```
red.help(4)
red.help(True)
```

You can read the whole documentation of `.help` here: *.help()*

7.1.3 Querying

Querying is inspired by BeautifulSoup. You have access to 2 methods: `.find` and `.find_all`, accepting the same arguments. The first one returns the first matched node and the second one returns the list of all the matched nodes.

The first argument is a string that represent the kind of the node you want to match on. The `identifiers` section displayed by the `.help()` method shows you several strings you can use to identify a kind of node. For example:

```
In [21]: red
Out[21]:
0   stuff = 1 + 2
1   print 'Beautiful result:', stuff
2   ' '

In [22]: red.help()
0 -----------------------------------------------------
AssignmentNode()
  # identifiers: assign, assignment, assignment_, assignmentnode
  operator=''
  target ->
    NameNode()
```

(continues on next page)

(continued from previous page)

```
      # identifiers: name, name_, namenode
      value='stuff'
  annotation ->
    None
  value ->
    BinaryOperatorNode()
      # identifiers: binary_operator, binary_operator_, binaryoperator,␣
↪binaryoperatornode
      value='+'
      first ->
        IntNode() ...
      second ->
        IntNode() ...
1 -----------------------------------------------------
EndlNode()
  # identifiers: endl, endl_, endlnode
  value='\n'
  indent=''
2 -----------------------------------------------------
PrintNode()
  # identifiers: print, print_, printnode
  destination ->
    None
  value ->
    * StringNode()
        # identifiers: string, string_, stringnode
        value="'Beautiful result:'"
    * NameNode()
        # identifiers: name, name_, namenode
        value='stuff'
3 -----------------------------------------------------
SpaceNode()
  # identifiers: space, space_, spacenode
  value=' '

In [23]: red.find("assignment")
Out[23]: stuff = 1 + 2

In [24]: red.find("print")
Out[24]: print 'Beautiful result:', stuff

In [25]: red.find_all("int")
Out[25]:
0   1
1   2
```

Then, you can pass as many keyword arguments as you want. They will filter the returned list on the attributes of the node and keep only those matching all attributes:

```
In [26]: red.find("int", value=2)
```

The only special argument you can pass is `recursive` that determine if the query is done recursively. By default it is set at `True`, just pass `recursive=False` to `.find` or `.find_all` to avoid that.

Queries are very powerful: you can pass lambdas, regexes, a short hand syntax for regexes and globs, a tuple of string instead of a string for the node kind, a global regex that receives the node (instead of a regex per attribute), etc. You can read all of that in the documentation: *Querying*.

Finally, `.find` and `.find_all` also have a shortcut syntax (exactly like in BeautifulSoup):

```
In [27]: red.find("int")
Out[27]: 1

In [28]: red.int
Out[28]: 1

In [29]: red.find_all("int", value=2)
Out[29]:

In [30]: red("int", value=2)
Out[30]:
```

But be aware that if you do a `red.something_that_can_be_a_node_identifier` and this is also not an attribute of a node, this will raise an `AttributeError`.

7.1.4 Modification

Nodes modification is extremely simple in RedBaron: you just have to set the attribute of the node you want to modify with a string containing python source code. Just look by yourself:

```
In [31]: red
Out[31]:
0   stuff = 1 + 2
1   print 'Beautiful result:', stuff
2   ' '

In [32]: red[0].target = "something_else"

In [33]: red[0].value = "42 * 34"

In [34]: red
Out[34]:
0   something_else = 42 * 34
1   print 'Beautiful result:', stuff
2   ' '

In [35]: red[1].value = "'Hello World!'"

In [36]: red
Out[36]:
0   something_else = 42 * 34
1   print 'Hello World!'
2   ' '
```

Notice that this also works with complex attributes like the body of a function. Here RedBaron makes a lot of effort to correctly format your input so you can pass it pretty much anything:

```
In [37]: red = RedBaron("def a():\n    pass")

In [38]: red[0].value = "1 + 1"

In [39]: red  # correctly indented
Out[39]:
0   def a():
```

(continues on next page)

(continued from previous page)

```
        1 + 1


In [40]: red[0].value = "\n\n\n           stuff\n"

In [41]: red  # again
Out[41]:
0   def a():


                   stuff
```

And this works too for more complex situations where the node is indented and followed by another node whose indentation can't be broken and other low level details that you don't want to hear about (but if you wish too, this is detailed in the full documentation).

And *voilà*, easy source code modification! You can also pass RedBaron node objects or Baron JSON FST that you have obtain is some way or another, for example by using `.copy()`:

```
In [42]: red = RedBaron("stuff = 1 + 2\nprint(stuff)")

In [43]: red
Out[43]:
0   stuff = 1 + 2
1   print(stuff)

In [44]: i = red[0].value.copy()

In [45]: red[1].value = i

In [46]: red
Out[46]:
0   stuff = 1 + 2
1   print1 + 2
```

You can also replace a node *in place* using the `.replace()` method. **Warning**: the `.replace()` expects that the string you pass represents a whole valid python program (so for example: `.replace("*args, **kwargs")` won't work). This limitation should be raised in the future.

```
In [47]: red
Out[47]:
0   stuff = 1 + 2
1   print1 + 2

In [48]: red[0].value.replace("1234")

In [49]: red
Out[49]:
0   stuff = 1234
1   print1 + 2
```

This is generally very useful when working on queries. For example (a real life example), here is the code to replace every `print stuff` (prints statement of **one** argument, an example with multiple arguments is left as an exercise to the reader) with `logger.debug(stuff)`:

```
red("print", value=lambda x: len(x) == 1).map(lambda x: x.replace("logger.debug(%s)"
↪% x.value.dumps()))
```

(`.map()` will be covered at the end of the tutorial but should speak for itself.)

You can read everything about modifications in RedBaron here: *Modifying*

7.1.5 Playing with list of nodes

The last big concept of RedBaron covered in this tutorial is how to handle list of nodes. The problem for short is that, for a python developer, the list `[1, 2, 3]` has 3 items but it has 5 items in the FST world, because it needs to take into account the commas. It is not sufficient to know that it is a comma separated list because each comma can have a different formatting. This is a pattern you find in every list of nodes, the separator being either commas, dots (eg: `a.b(c)[d]`) or end of line characters (for lines of code).

Having to deal with those separators is extremely annoying and error prone, so, RedBaron offers an abstraction that hides all this for you! You just have to deal with those list of nodes like if they were regular python list and everything will be fine. See by yourself:

```
In [50]: red = RedBaron("[1, 2, 3]")

In [51]: red.help()
0 -----------------------------------------------------
ListNode()
  # identifiers: list, list_, listnode
  value ->
    * IntNode()
        # identifiers: int, int_, intnode
        value='1'
    * IntNode()
        # identifiers: int, int_, intnode
        value='2'
    * IntNode()
        # identifiers: int, int_, intnode
        value='3'

In [52]: red[0].value  # see: no explicit commas to deal with
Out[52]:
0   1
1   2
2   3

In [53]: red[0].value.append("4")

In [54]: red  # comma has been added for us
Out[54]: 0   [1, 2, 3, 4]
```

This abstraction is called a proxy list. They can even detect indentation style for comma separated lists:

```
In [55]: red = RedBaron("[\n    1,\n    2,\n    3,\n]")

In [56]: red
Out[56]:
0   [
        1,
        2,
        3,
    ]

In [57]: red[0].value.append("caramba")
```

(continues on next page)

(continued from previous page)

```
In [58]: red
Out[58]:
0   [
        1,
        2,
        3,
        caramba,
    ]
```

This also work with nodes separated by dots:

```
In [59]: red = RedBaron("a.b(c)[d]")

In [60]: red
Out[60]: 0   a.b(c)[d]

In [61]: red[0].value.extend(["e", "(f)", "[g:h]"])

In [62]: red
Out[62]: 0   a.b(c)[d].e(f)[g:h]
```

And lines of code (note that the blank lines are explicitly shown and it is intended as such, see the documentation for more information: *Proxy List*):

```
In [63]: red = RedBaron("a = 1\n\nprint(a)")

In [64]: red
Out[64]:
0   a = 1
1   '\n'
2   print(a)

In [65]: red.insert(1, "if a:\n    print('a == 1')")

In [66]: red
Out[66]:
0   a = 1
1   if a:
        print('a == 1')

2   '\n'
3   print(a)
```

The important things to remember are that:

- Every method and protocol of python lists (except `sort` and `reversed`) works on proxy list.
- And every node list in python is wrapped by a proxy list.

The raw list is stored on the `.node_list` attribute of the proxy list:

```
In [67]: red = RedBaron("[1, 2, 3]")

In [68]: red[0].node_list
Out[68]:
0   1
```

(continues on next page)

(continued from previous page)

```
1    ,
2    2
3    ,
4    3
```

Warning: the proxyfied list and the proxy list are only synced from the proxy list to the raw list. If you start to modify the raw list, don't use the proxy list anymore or you'll have strange bugs! This might change in the future.

One last thing: if the proxy list is stored on the `.value` attribute, you can directly call the methods on the holder node. This is done because it is more intuitive, see by yourself:

```
red = RedBaron("[1, 2, 3]")

red[0].append("4")  # is exactly the same than the next line
red[0].value.append("4")
```

7.1.6 Misc things

A short list of useful features of RedBaron:

- `.map`, a method of RedBaron lists that takes a callable (like a lambda or a function), apply it to every one of its members and returns a RedBaron list containing the result of the call
- `.apply` same than `.map` except it returns a RedBaron list of the nodes on which the callable has been applied (i.e. the members before the call instead of the members after the call) (for simplicity we uses the `int` builtin function here, you might want to look at `to_python` in the future for a more generic conversion operation)

```
In [69]: red = RedBaron("[1, 2, 3]")

In [70]: red("int").map(lambda x: int(x.value) + 42)
Out[70]:
0    43
1    44
2    45

In [71]: red("int").apply(lambda x: int(x.value) + 42)
Out[71]:
0    1
1    2
2    3
```

- `.filter`, another method of RedBaron list, it takes a callable and return a RedBaron list containing the nodes for which the callable has returned True (or something that is tested has True in python)

```
In [72]: red = RedBaron("[1, 2, 3]")

In [73]: red("int").filter(lambda x: int(x.value) % 2 == 1)  # odd numbers
Out[73]:
0    1
1    3
```

- `.next` gives the node just after the current one if the node is in a list
- `.previous` does the inverse
- `.parent` gives the holder of this node

```
In [74]: red = RedBaron("[1, 2, 3]")

In [75]: red.int_
Out[75]: 1

In [76]: red.int_.next
Out[76]: ,

In [77]: red.int_.previous  # None because nothing is behind it

In [78]: red.int_.parent
Out[78]: [1, 2, 3]
```

And you can find all the others various RedBaron features here: *Other*

7.2 Why is this important?

The usage of an FST might not be obvious at first sight so let's consider a series of problems to illustrate it. Let's say that you want to write a program that will:

- rename a variable in a source file... without clashing with things that are not a variable (example: stuff inside a string)
- inline a function/method
- extract a function/method from a series of line of code
- split a class into several classes
- split a file into several modules
- convert your whole code base from one ORM to another
- do custom refactoring operation not implemented by IDE/rope
- implement the class browser of smalltalk for python (the whole one where you can edit the code of the methods, not just showing code)

It is very likely that you will end up with the awkward feeling of writing clumsy weak code that is very likely to break because you didn't thought about all the annoying special cases and the formatting keeps bothering you. You may end up playing with ast.py until you realize that it removes too much information to be suitable for those situations. You will probably ditch this task as simple too complicated and really not worth the effort. You are missing a good abstraction that will take care of all of the code structure and formatting for you so you can concentrate on your task.

The FST tries to be this abstraction. With it you can now work on a tree which represents your code with its formatting. Moreover, since it is the exact representation of your code, modifying it and converting it back to a string will give you back your code only modified where you have modified the tree.

Said in another way, what I'm trying to achieve with Baron is a paradigm change in which writing code that will modify code is now a realistic task that is worth the price (I'm not saying a simple task, but a realistic task, it's still a complex task).

7.2.1 Other usages

Having an FST (or at least a good abstraction build on it) also makes it easier to do code generation and code analysis while those two operations are already quite feasible (using ast.py and a templating engine for example).

Next

To start playing with RedBaron read *Basics*.

7.3 Basics

RedBaron is very simple to use, you just need to import it and feed him with a string:

```
from redbaron import RedBaron

red = RedBaron("print('hello world!')")
```

But what you should be really doing is using RedBaron directly into a shell (I recommend IPython but bpython is cool too), it has been thought for it, like BeautifulSoup.

```
In [1]: from redbaron import RedBaron

In [2]: red = RedBaron("hello = 'Hello World!'\nprint(hello)")

In [3]: red
Out[3]:
0   hello = 'Hello World!'
1   print(hello)
```

As you can see, when displayed in a shell, a RedBaron instance renders to the actual content so you easily see what you are doing when playing interactively with it (just like a BeautifulSoup instance).

There are 2 families of Node in RedBaron: NodeList and standalone Node. Since a Python program is a list of operations, RedBaron will always be a list. This is why when displayed you see integers on the left, those are the index in the list of the nodes of the right, so as expected:

```
In [4]: red[1]
Out[4]: print(hello)
```

You get the *print* Node that was located at 2. As you can see, here we are on a standalone Node, so we don't get the list of indexes of the left.

7.3.1 .help()

Another useful function is `.help()`. It displays the RedBaron nodes tree helping you understand how is it composed and how you can use it:

```
In [5]: red[0]
Out[5]: hello = 'Hello World!'

In [6]: red[0].help()
AssignmentNode()
  # identifiers: assign, assignment, assignment_, assignmentnode
  operator=''
  target ->
    NameNode()
      # identifiers: name, name_, namenode
      value='hello'
  annotation ->
```

(continues on next page)

(continued from previous page)

```
    None
  value ->
    StringNode()
      # identifiers: string, string_, stringnode
      value="'Hello World!'"
```

Here, as you can see, `hello = 'Hello World!'` is an `AssignmentNode` and it has 2 attributes: `target` and `value`. Those 2 attributes are 2 other nodes, a `NameNode` for the variable `hello` and a `StringNode` for the string. Those 2 nodes each have one attribute `value` that is their content.

One rule with Baron: **every node has a value attribute** that contains its value (in case of a node with multiple data, `value` points to the most obvious one, for example, in a function definition it's the body of the function). The **only exceptions** are nodes where it doesn't make any sense, for example a `PassNode` (representing the keyword `pass`) simply doesn't contain anything.

Like the `repr`, `.help()` has also a display showing index number when called on a `NodeList`:

```
In [7]: red.help()
0 -----------------------------------------------------
AssignmentNode()
  # identifiers: assign, assignment, assignment_, assignmentnode
  operator=''
  target ->
    NameNode()
      # identifiers: name, name_, namenode
      value='hello'
  annotation ->
    None
  value ->
    StringNode()
      # identifiers: string, string_, stringnode
      value="'Hello World!'"
1 -----------------------------------------------------
EndlNode()
  # identifiers: endl, endl_, endlnode
  value='\n'
  indent=''
2 -----------------------------------------------------
PrintNode()
  # identifiers: print, print_, printnode
  destination ->
    None
  value ->
    * AssociativeParenthesisNode()
        # identifiers: associative_parenthesis, associative_parenthesis_,␣
↪associativeparenthesis, associativeparenthesisnode
        value ->
          NameNode() ...
```

The best way to understand how `.help()` works is to remember that RedBaron is mapping from Baron FST which is JSON. This means that RedBaron node can be composed of either: string, bool, numbers, list or other nodes and the key are always string.

helpers

Some nodes come with helpers method, `.help()` displays them when they are present:

```
In [8]: red = RedBaron("import a, b, c as d")

In [9]: red.help(deep=1)
0 -----------------------------------------------------
ImportNode()
  # identifiers: import, import_, importnode
  # helpers: modules, names
  value ->
    * DottedAsNameNode() ...
    * DottedAsNameNode() ...
    * DottedAsNameNode() ...
```

You can read their documentation using the `?` magic of ipython:

```
In [10]: print(red[0].names.__doc__)  # you can do "red[0].names?" in IPython shell
return a list of string of new names inserted in the python context

In [11]: red[0].names()
Out[11]: ['a', 'b', 'd']

In [12]: print(red[0].modules.__doc__)
return a list of string of modules imported

In [13]: red[0].modules()
Out[13]: ['a', 'b', 'c']
```

If you come with cool helpers, don't hesitate to propose them in a pull request!

deep

`.help()` accept a deep argument on how far in the tree it should show the `.help()` of subnode. By default its value is `2`. You can pass the value `True` if you want to display the whole tree.

```
In [14]: red = RedBaron("a = b if c else d")

In [15]: red.help()
0 -----------------------------------------------------
AssignmentNode()
  # identifiers: assign, assignment, assignment_, assignmentnode
  operator=''
  target ->
    NameNode()
      # identifiers: name, name_, namenode
      value='a'
  annotation ->
    None
  value ->
    TernaryOperatorNode()
      # identifiers: ternary_operator, ternary_operator_, ternaryoperator,␣
↪ternaryoperatornode
      first ->
        NameNode() ...
      value ->
        NameNode() ...
      second ->
        NameNode() ...
```

(continues on next page)

(continued from previous page)

```
In [16]: red.help(0)
0 -----------------------------------------------------
AssignmentNode() ...

In [17]: red.help(deep=1)  # you can name the argument too
0 -----------------------------------------------------
AssignmentNode()
  # identifiers: assign, assignment, assignment_, assignmentnode
  operator=''
  target ->
    NameNode() ...
  annotation ->
    None
  value ->
    TernaryOperatorNode() ...

In [18]: red.help(True)
0 -----------------------------------------------------
AssignmentNode()
  # identifiers: assign, assignment, assignment_, assignmentnode
  operator=''
  target ->
    NameNode()
      # identifiers: name, name_, namenode
      value='a'
  annotation ->
    None
  value ->
    TernaryOperatorNode()
      # identifiers: ternary_operator, ternary_operator_, ternaryoperator,␣
↪ternaryoperatornode
      first ->
        NameNode()
          # identifiers: name, name_, namenode
          value='b'
      value ->
        NameNode()
          # identifiers: name, name_, namenode
          value='c'
      second ->
        NameNode()
          # identifiers: name, name_, namenode
          value='d'
```

with_formatting

`.help()` accepts the option `with_formatting` that is set at `False` by default. If set at `True` it will also display the attributes responsible for holding the formatting of the node (they are always node list):

```
In [19]: red.help(with_formatting=True)
0 -----------------------------------------------------
AssignmentNode()
  # identifiers: assign, assignment, assignment_, assignmentnode
  operator=''
```

(continues on next page)

(continued from previous page)

```
  target ->
    NameNode()
      # identifiers: name, name_, namenode
      value='a'
  annotation ->
    None
  value ->
    TernaryOperatorNode()
      # identifiers: ternary_operator, ternary_operator_, ternaryoperator,␣
↪ternaryoperatornode
      first ->
        NameNode() ...
      value ->
        NameNode() ...
      second ->
        NameNode() ...
      first_formatting ->
        * SpaceNode() ...
      second_formatting ->
        * SpaceNode() ...
      third_formatting ->
        * SpaceNode() ...
      fourth_formatting ->
        * SpaceNode() ...
  annotation_first_formatting ->
  annotation_second_formatting ->
  first_formatting ->
    * SpaceNode()
        # identifiers: space, space_, spacenode
        value=' '
  second_formatting ->
    * SpaceNode()
        # identifiers: space, space_, spacenode
        value=' '
```

Those attributes are always surrounding syntax element of Python like `[] () , . {}` or keywords. You should, normally, not have a lot of reasons to play with them. You can find a detailed version of each nodes here: *Nodes References Page*.

7.3.2 nodes structure

Nodes can have 3 kind of attributes (which can be accessed like normal object attributes):

- data attributes, which are nearly always strings. They are shown with a = in `.help()`. `.value` here for example.

```
In [20]: red = RedBaron("variable")

In [21]: red[0].help()
NameNode()
  # identifiers: name, name_, namenode
  value='variable'

In [22]: red[0].value
Out[22]: 'variable'
```

- node attributes, which are other nodes. They are shown with a `->` followed by the name of the other node at the next line in `.help()`. `.target` and `.value` here for example.

```
In [23]: red = RedBaron("a = 1")

In [24]: red[0].help()
AssignmentNode()
  # identifiers: assign, assignment, assignment_, assignmentnode
  operator=''
  target ->
    NameNode()
      # identifiers: name, name_, namenode
      value='a'
  annotation ->
    None
  value ->
    IntNode()
      # identifiers: int, int_, intnode
      value='1'

In [25]: red[0].target.help()
NameNode()
  # identifiers: name, name_, namenode
  value='a'
```

- nodelist attributes, which are lists of other nodes. They are shown with a `->` followed by a series of names of the other nodes starting with a `*` for every item of the list. `.value` here for example:

```
In [26]: red = RedBaron("[1, 2, 3]")

In [27]: red[0].help()
ListNode()
  # identifiers: list, list_, listnode
  value ->
    * IntNode()
        # identifiers: int, int_, intnode
        value='1'
    * IntNode()
        # identifiers: int, int_, intnode
        value='2'
    * IntNode()
        # identifiers: int, int_, intnode
        value='3'

In [28]: red[0].value[0].help()
IntNode()
  # identifiers: int, int_, intnode
  value='1'
```

7.3.3 .dumps(), transform the tree into source code

To transform a RedBaron tree back into source code, use the `.dumps()` method. This will transform the **current selection** back into code.

```
In [29]: red = RedBaron("a = 1")
```

(continues on next page)

(continued from previous page)

```
In [30]: red.dumps()
Out[30]: 'a = 1'

In [31]: red[0].target.dumps()
Out[31]: 'a'
```

7.3.4 .fst(), transform the RedBaron tree into Baron FST

To transform a RedBaron tree into Baron Full Syntax Tree, just use the `.fst()` method. This will transform the **current selection** into FST.

```
In [32]: red = RedBaron("a = 1")

In [33]: red.fst()
Out[33]:
[{'annotation': {},
  'annotation_first_formatting': [],
  'annotation_second_formatting': [],
  '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 [34]: red[0].target.fst()
Out[34]: {'type': 'name', 'value': 'a'}
```

While I don't see a lot of occasions where you might need this, this will allow you to better understand how Baron and RedBaron are working.

7.3.5 .copy()

If you want to copy a RedBaron node you can use the `.copy()` method this way:

```
In [35]: red = RedBaron("a = b")

In [36]: red[0].target.copy()
Out[36]: a
```

Next

To learn how to find things in RedBaron read *Querying*.

7.4 Querying

As you have seen in the previous section, you can navigate into RedBaron tree only using attribute access and index access on list of nodes with the use of the `.help()` method to know what you can do. However, RedBaron offers way more powerful and convenient tools to do that.

7.4.1 .find()

To retrieve a single node, you can use the `.find()` method by passing it one of the identifiers listed in `.help()` of node you want to get, this way:

```
In [1]: red = RedBaron("a = 1")

In [2]: red.help()
0 -----------------------------------------------------
AssignmentNode()
  # identifiers: assign, assignment, assignment_, assignmentnode
  operator=''
  target ->
    NameNode()
      # identifiers: name, name_, namenode
      value='a'
  annotation ->
    None
  value ->
    IntNode()
      # identifiers: int, int_, intnode
      value='1'

In [3]: red.find('NameNode').help()
NameNode()
  # identifiers: name, name_, namenode
  value='a'

In [4]: red.find('namenode').help()  # identifiers are not case sensitive
NameNode()
  # identifiers: name, name_, namenode
  value='a'

In [5]: red.find('name')
Out[5]: a
```

This will recursively travel the tree and return the first node of that type.

You can also specify attributes of the node that you want to match:

```
In [6]: red = RedBaron("a = b")

In [7]: red.find('name').help()
NameNode()
  # identifiers: name, name_, namenode
  value='a'

In [8]: red.find('name', value='b').help()
NameNode()
  # identifiers: name, name_, namenode
  value='b'
```

If you don't want a recursive approach but only on the first level on the current node or node list, you can pass `recursive=False` to `.find()`.

Like BeautifulSoup, RedBaron provides a shorthand to `.find()`, you can write the name of the target as an attribute of the node and this will do a `.find()` in the same fashion:

```
In [9]: red = RedBaron("a = b")

In [10]: red.find('name')
Out[10]: a

In [11]: red.name
Out[11]: a
```

You might have noticed that some identifiers end with a _, those are for the case where the identifier might be a Python reserved keyword like `if`, or `while` for example.

Be aware that if you do a `red.something_that_can_be_a_node_identifier` and this is also not an attribute of a node, this will raise an `AttributeError`.

7.4.2 .find_all()

`.find_all()` is extremely similar to `.find()` except it returns a node list containing all the matching queries instead of a single one. Like in BeautifulSoup, `__call__` is aliased to `find_all` (meaning that if you try to *call* the node this way `node(some_arguments)` this will call `.find_all()` with the arguments).

```
In [12]: red = RedBaron("a = b")

In [13]: red.find_all("NameNode")
Out[13]:
0   a
1   b

In [14]: red.find_all("name")
Out[14]:
0   a
1   b

In [15]: red.findAll("name")
Out[15]:
0   a
1   b

In [16]: red.findAll("name", value="b")
Out[16]: 0   b

In [17]: red("name", value="b")
Out[17]: 0   b
```

`.find_all()` also supports the option `recursive=False`.

7.4.3 Advanced querying

`.find()` and `.find_all()` offer more powerful comparison mean than just equality comparison.

Callable (lambda)

Instead of passing a string to test properties of the identifier of a node, you can pass a callable, like a lambda. It will receive the value as first argument:

```
In [18]: red = RedBaron("a = [1, 2, 3, 4]")

In [19]: red.find("int", value=lambda value: int(value) % 2 == 0)
Out[19]: 2

In [20]: red.find_all("int", value=lambda value: int(value) % 2 == 0)
Out[20]:
0   2
1   4

In [21]: red.find(lambda identifier: identifier == "comma")
Out[21]: ,

In [22]: red.find_all(lambda identifier: identifier == "comma")
Out[22]:
0   ,
1   ,
2   ,
```

Regex

Instead of passing a string to test properties of a node, you can pass a compiled regex:

```
In [23]: import re

In [24]: red = RedBaron("abcd = plop + pouf")

In [25]: red.find("name", value=re.compile("^p"))
Out[25]: plop

In [26]: red.find_all("name", value=re.compile("^p"))
Out[26]:
0   plop
1   pouf

In [27]: red.find(re.compile("^n"))
Out[27]: abcd

In [28]: red.find_all(re.compile("^n"))
Out[28]:
0   abcd
1   plop
2   pouf
```

Having to compile regex is boring, so you can use this shorthand syntax instead (prefixing a string with "re:"):

```
In [29]: red = RedBaron("abcd = plop + pouf")

In [30]: red.find("name", value="re:^p")
Out[30]: plop

In [31]: red.find_all("name", value="re:^p")
Out[31]:
0   plop
1   pouf
```

(continues on next page)

(continued from previous page)

```
In [32]: red.find("re:^n")
Out[32]: abcd

In [33]: red.find_all("re:^n")
Out[33]:
0   abcd
1   plop
2   pouf
```

Globs

Same than in a shell, you can use globs by prefixing the string with "g:":

```
In [34]: red = RedBaron("abcd = plop + pouf")

In [35]: red.find("name", value="g:p*")
Out[35]: plop

In [36]: red.find_all("name", value="g:p*")
Out[36]:
0   plop
1   pouf

In [37]: red.find("g:n*")
Out[37]: abcd

In [38]: red.find_all("g:n*")
Out[38]:
0   abcd
1   plop
2   pouf
```

In the background, the comparison is done using the fnmatch module of the standard lib.

List or tuple

You can pass a list as a shorthand to test if the tested attribute is in any of the member of the list/tuple:

```
In [39]: red = RedBaron("foo\nbar\nbaz")

In [40]: red.find("name", value=["foo", "baz"])
Out[40]: foo

In [41]: red.find("name", value=("foo", "baz"))
Out[41]: foo

In [42]: red("name", value=["foo", "baz"])
Out[42]:
0   foo
1   baz

In [43]: red("name", value=("foo", "baz"))
Out[43]:
```

(continues on next page)

(continued from previous page)

```
0   foo
1   baz
```

```
In [44]: red = RedBaron("1\nstuff\n'string'\n")

In [45]: red.find(["int", "string"])
Out[45]: 1

In [46]: red(["int", "string"])
Out[46]:
0   1
1   'string'
```

*args and default value

You can also pass as many callable as args (without giving it a key) as you want, those callables will receive the node itself as first argument (and must return a value that will be tested as a boolable):

```
In [47]: red = RedBaron("a = [1, 2, 3, 4]")

In [48]: red.find("int", lambda node: int(node.value) % 2 == 0)
Out[48]: 2

In [49]: red.find_all("int", lambda node: int(node.value) % 2 == 0)
Out[49]:
0   2
1   4

In [50]: red.find("int", lambda node: int(node.value) % 2 == 0, lambda node: int(node.
↪value) == 4)
Out[50]: 4
```

To ease the usage of RedBaron in ipython (and in general), you can pass any of the previous testing methods (**except the lambda**) as the **first** argument of *args, it will be tested against the default testing attribute which is the "value" attribute by default. This mean that: `red.find("name", "foo")` is the equivalent of `red.find("name", value="foo")`.

If the default tested attribute is different, it will be shown in `.help()`. For now, the 2 only cases where this happens is on class node and funcdef node where the attribute is "name".

```
In [51]: red = RedBaron("foo\ndef bar(): pass\nbaz\ndef badger(): pass")

In [52]: red.find("name", "baz")
Out[52]: baz

In [53]: red.find("def", "bar")
Out[53]: def bar(): pass

In [54]: red.find("def").help()
DefNode()
  # identifiers: def, def_, defnode, funcdef, funcdef_
  # default test value: name
  async=False
  name='bar'
  return_annotation ->
```

(continues on next page)

(continued from previous page)

```
    None
  decorators ->
  arguments ->
  value ->
    * PassNode()
        # identifiers: pass, pass_, passnode
```

Next

To learn how to modify stuff in RedBaron read *Modifying*.

7.5 Modifying

7.5.1 Principle

When it comes to modifying the tree, the normal classical way would tell you to use the RedBaron nodes constructors, like this:

```
In [1]: from redbaron import RedBaron, NameNode

In [2]: red = RedBaron("a = 1")

In [3]: red[0].value
Out[3]: 1

In [4]: red[0].value = NameNode({'first_formatting': [{'type': 'space', 'value': ' '}
↪], 'value': '+', 'second_formatting': [{'type': 'space', 'value': ' '}], 'second': {
↪'section': 'number', 'type': 'int', 'value': '1'}, 'type': 'binary_operator', 'first
↪': {'section': 'number', 'type': 'int', 'value': '1'}})

In [5]: red
Out[5]: 0   a = 1 + 1
```

As you can see, this is totally impracticable. So, to solve this problem, RedBaron adopt a simple logic: you already know how to code in python, so, just send python code in form of a string, RedBaron will takes care or parsing and injecting it into its tree. This give an extremely simple and intuitive API:

```
In [6]: red = RedBaron("a = 1")

In [7]: red[0].value
Out[7]: 1

In [8]: red[0].value = "1 + 1"

In [9]: red
Out[9]: 0   a = 1 + 1
```

The details on how you can modify **every** nodes can be found here: *Nodes References Page*.

7.5.2 Code block modifications

The modification of python code block (like the body of a function or a while loop) is also possible this way. RedBaron will takes care for you or formatting you input the right way (adding surrounding blank lines and settings the correct indentation for the every line).

Example:

```
In [10]: red = RedBaron("while True: pass")

In [11]: red[0].value = "plop"

In [12]: red
Out[12]:
0   while True:
        plop


In [13]: red[0].value = "                              this_will_be_correctly_indented"

In [14]: red
Out[14]:
0   while True:
        this_will_be_correctly_indented
```

You have the full list of cases handled on this page: *Nodes References Page*.

7.5.3 Details

As you might have already noticed, you can set attributes of a node with a string or a RedBaron node. This is also possible by directly passing FST.

Here is an IPython session illustrating all the possibilities (be sure to have read the "node structures" in basics to understand what is happening):

```
In [15]: from redbaron import RedBaron

In [16]: red = RedBaron("a = b")
```

Data attribute, no parsing

```
In [17]: red.name.help()
NameNode()
  # identifiers: name, name_, namenode
  value='a'

In [18]: red.name.value = "something_else"

In [19]: red
Out[19]: 0   something_else = b
```

Node attribute with a string: parsing with RedBaron

```
In [20]: red[0].help()
AssignmentNode()
  # identifiers: assign, assignment, assignment_, assignmentnode
  operator=''
  target ->
    NameNode()
      # identifiers: name, name_, namenode
      value='something_else'
  annotation ->
    None
  value ->
    NameNode()
      # identifiers: name, name_, namenode
      value='b'

In [21]: red[0].value = "42 * pouet"

In [22]: red
Out[22]: 0   something_else = 42 * pouet
```

Node attribute with FST data: transformation into RedBaron objects

```
In [23]: red[0].value = {"type": "name", "value": "pouet"}

In [24]: red
Out[24]: 0   something_else = pouet
```

List attribute with a string: parsing with RedBaron

```
In [25]: red = RedBaron("[1, 2, 3]")

In [26]: red[0].help()
ListNode()
  # identifiers: list, list_, listnode
  value ->
    * IntNode()
        # identifiers: int, int_, intnode
        value='1'
    * IntNode()
        # identifiers: int, int_, intnode
        value='2'
    * IntNode()
        # identifiers: int, int_, intnode
        value='3'

In [27]: red[0].value = "caramba"

In [28]: red
Out[28]: 0   [caramba]

In [29]: red[0].value = "4, 5, 6"
```

(continues on next page)

(continued from previous page)

```
In [30]: red
Out[30]: 0   [4, 5, 6]
```

List node attribute with FST: transformation into RedBaron objects

```
In [31]: red[0].value = {"type": "name", "value": "pouet"}

In [32]: red
Out[32]: 0   [pouet]

In [33]: red[0].value = [{"type": "name", "value": "pouet"}]

In [34]: red
Out[34]: 0   [pouet]
```

List node attribute with mixed content: parsing/transformation depending of the context

```
In [35]: red[0].value = [{"type": "name", "value": "pouet"}, {"type": "comma", "first_
↪formatting": [], "second_formatting": []}, "pouet ,", NameNode({"type": "name",
↪"value": "plop"})]

In [36]: red
Out[36]: 0   [pouet,pouet ,plop]
```

7.5.4 Auto assignment of .parent and .on_attribute

When you modify an attribute of a node or a node list, RedBaron will take care of setting the `.parent` value of the new attribute to the corresponding node.

This will be done if you set the attribute value using either a `string`, a `fst node`, an instance of a node or a node list.

The same is done for `.on_attribute`.

Next

To learn how to work with list of things in RedBaron read *Proxy List*.

7.6 Proxy List

7.6.1 Problem

For a python developer, the list `[1, 2, 3]` has 3 members, which is true in the python world, but in the "source code modification" world, this list has 5 elements because you have to count the 2 commas. Indeed each comma needs to be taken into account separately because they can have a different formatting.

This makes things quite annoying to deal with because you have to think about the formatting too! For example, if you want to append an item to a list, you need to take care of a lot of details:

- if the list is empty you don't have to put a comma
- otherwise yes
- but wait, what happens if there is a trailing comma?
- also, what to do if the list is declared in an indented way (with `"\n    "` after every comma for example)?
- etc...

And that's only for a comma separated list of things: you also have the same formatting details to care about for dot separated lists (e.g. `a.b.c().d[plop]`) and endl separated lists (a python code block, or you whole source file).

You don't want to have to deal with this.

7.6.2 Solution

To avoid you to deal with all this boring low level details, RedBaron implements "proxy lists". This abstraction gives you the impression that the list of things you are dealing with behave the same way than in the python world while taking care of all the low level formatting details.

The "proxy lists" has the same API than a python list so they should be really intuitive to use.

For example:

```
In [1]: red = RedBaron("[1, 2, 3]")

In [2]: red[0].value.append("42")

In [3]: red
Out[3]: 0   [1, 2, 3, 42]

In [4]: del red[0].value[2]

In [5]: red
Out[5]: 0   [1, 2, 42]
```

There are, for now, 4 kind of proxy lists:

- `CommaProxyList` which handles comma separated lists
- `DotProxyList` which handles `atomtrailers` (those kind of constructions: `a.b[plop].c()`)
- `LineProxyList` which handles lines of code (like the body of a function or the whole source code)
- `DecoratorLineProxyList` which handles lists of decorators (they are nearly the same as `LineProxyList`)

Be aware that the proxy list are set on the attribute that is a list, not on the node holding the list. See the 'value' attribute access in the examples below.

7.6.3 Usage

As said, proxy lists have the exact same API than python lists (at the exception that they don't implement the `sort` and `reverse` methods). Every method accepts as input the same inputs that you can use to modify a node in RedBaron. This means that you can pass a string containing source code, an FST or a RedBaron node.

Here is a session demonstrating every method of a proxy list:

```
In [6]: red = RedBaron("[1, 2, 3]")
```

Please refer to python list documentation if you want to know the exact behavior or those methods (or send a patch to improve this documentation).

append

```
In [7]: red
Out[7]: 0   [1, 2, 3]

In [8]: red[0].value.append("plop")

In [9]: red
Out[9]: 0   [1, 2, 3, plop]

In [10]: red[0].value
Out[10]:
0   1
1   2
2   3
3   plop
```

insert

```
In [11]: red
Out[11]: 0   [1, 2, 3, plop]

In [12]: red[0].value.insert(1, "42")

In [13]: red
Out[13]: 0   [1, 42, 2, 3, plop]

In [14]: red[0].value
Out[14]:
0   1
1   42
2   2
3   3
4   plop
```

extend

```
In [15]: red
Out[15]: 0   [1, 42, 2, 3, plop]

In [16]: red[0].value.extend(["pif", "paf", "pouf"])

In [17]: red
Out[17]: 0   [1, 42, 2, 3, plop, pif, paf, pouf]

In [18]: red[0].value
Out[18]:
```

(continues on next page)

(continued from previous page)

```
0   1
1   42
2   2
3   3
4   plop
5   pif
6   paf
7   pouf
```

pop

```
In [19]: red
Out[19]: 0   [1, 42, 2, 3, plop, pif, paf, pouf]

In [20]: red[0].value.pop()

In [21]: red
Out[21]: 0   [1, 42, 2, 3, plop, pif, paf]

In [22]: red[0].value
Out[22]:
0   1
1   42
2   2
3   3
4   plop
5   pif
6   paf

In [23]: red[0].value.pop(3)

In [24]: red
Out[24]: 0   [1, 42, 2, plop, pif, paf]

In [25]: red[0].value
Out[25]:
0   1
1   42
2   2
3   plop
4   pif
5   paf
```

__getitem__

```
In [26]: red
Out[26]: 0   [1, 42, 2, plop, pif, paf]

In [27]: red[0].value
Out[27]:
0   1
1   42
2   2
```

(continues on next page)

(continued from previous page)

```
3   plop
4   pif
5   paf

In [28]: red[0].value[2]
Out[28]: 2
```

__setitem__

```
In [29]: red
Out[29]: 0   [1, 42, 2, plop, pif, paf]

In [30]: red[0].value[2] = "1 + 1"

In [31]: red
Out[31]: 0   [1, 42, 1 + 1, plop, pif, paf]

In [32]: red[0].value
Out[32]:
0   1
1   42
2   1 + 1
3   plop
4   pif
5   paf
```

remove

```
In [33]: red
Out[33]: 0   [1, 42, 1 + 1, plop, pif, paf]

In [34]: red[0].value.remove(red[0].value[2])

In [35]: red
Out[35]: 0   [1, 42, plop, pif, paf]

In [36]: red[0].value
Out[36]:
0   1
1   42
2   plop
3   pif
4   paf
```

index

```
In [37]: red
Out[37]: 0   [1, 42, plop, pif, paf]

In [38]: red[0].value
Out[38]:
```

(continues on next page)

(continued from previous page)

```
0    1
1    42
2    plop
3    pif
4    paf

In [39]: red[0].value.index(red[0].value[2])
Out[39]: 2
```

count

```
In [40]: red
Out[40]: 0    [1, 42, plop, pif, paf]

In [41]: red[0].value
Out[41]:
0    1
1    42
2    plop
3    pif
4    paf

In [42]: red[0].value.count(red[0].value[2])
Out[42]: 1
```

len

```
In [43]: red
Out[43]: 0    [1, 42, plop, pif, paf]

In [44]: red[0].value
Out[44]:
0    1
1    42
2    plop
3    pif
4    paf

In [45]: len(red[0].value)
Out[45]: 5
```

__delitem__

```
In [46]: red
Out[46]: 0    [1, 42, plop, pif, paf]

In [47]: del red[0].value[2]

In [48]: red
Out[48]: 0    [1, 42, pif, paf]
```

(continues on next page)

(continued from previous page)

```
In [49]: red[0].value
Out[49]:
0   1
1   42
2   pif
3   paf
```

in

```
In [50]: red
Out[50]: 0   [1, 42, pif, paf]

In [51]: red[0].value[2] in red[0].value
Out[51]: False
```

__iter__

```
In [52]: red
Out[52]: 0   [1, 42, pif, paf]

In [53]: for i in red[0].value:
   ....:     print(i.dumps())
   ....:
1
42
pif
paf
```

__getslice__

```
In [54]: red
Out[54]: 0   [1, 42, pif, paf]

In [55]: red[0].value
Out[55]:
0   1
1   42
2   pif
3   paf

In [56]: red[0].value[2:4]
Out[56]:
0   pif
1   paf
```

__setslice__

```
In [57]: red
Out[57]: 0   [1, 42, pif, paf]
```

(continues on next page)

(continued from previous page)

```
In [58]: red[0].value[2:4] = ["1 + 1", "a", "b", "c"]

In [59]: red
Out[59]: 0   [1, 42, 1 + 1, a, b, c]

In [60]: red[0].value
Out[60]:
0   1
1   42
2   1 + 1
3   a
4   b
5   c
```

__delslice__

```
In [61]: red
Out[61]: 0   [1, 42, 1 + 1, a, b, c]

In [62]: red[0].value[2:5]
Out[62]:
0   1 + 1
1   a
2   b

In [63]: del red[0].value[2:5]

In [64]: red
Out[64]: 0   [1, 42, c]

In [65]: red[0].value
Out[65]:
0   1
1   42
2   c
```

7.6.4 Access the unproxified node list

The unproxified node list is stored under the attribute `node_list` of the proxy list. **Be aware that, for now, the proxy won't detect if you directly modify the unproxified node list, this will cause bugs if you modify the unproxified list then use the proxy list directly**. So, for now, only use one or the other.

```
In [66]: red = RedBaron("[1, 2, 3]")

In [67]: red[0].value.node_list
Out[67]:
0   1
1   ,
2   2
3   ,
4   3
```

(continues on next page)

(continued from previous page)

```
In [68]: red[0].value
Out[68]:
0    1
1    2
2    3
```

7.6.5 Omitting ".value"

For convenience, and because this is a super common typo error, if a node has a proxy list on its `.value` attribute, you can omit to access it and the method access will be automatically redirect to it.

This means that the 2 next lines are equivalent:

```
In [69]: red[0]
Out[69]: [1, 2, 3]

In [70]: red[0].value.append("plop")

In [71]: red[0].append("plop")
```

7.6.6 CommaProxyList

CommaProxyList is the most generic and most obvious proxy list, all the examples above are made using it.

It is used everywhere where values are separated by commas.

7.6.7 DotProxyList

DotProxyList is nearly as generic as the CommaProxyList. The specific case of a DotProxyList is that it is intelligent enough to not add a "." before a "call" (`(a, b=c, *d, **e)`) or a "getitem" (`[foobar]`).

```
In [72]: red = RedBaron("a.b(c).d[e]")

In [73]: red[0].value
Out[73]:
0    a
1    b
2    (c)
3    d
4    [e]

In [74]: red[0].extend(["[stuff]", "f", "(g, h)"])

In [75]: red[0]
Out[75]: a.b(c).d[e][stuff].f(g, h)

In [76]: red[0].value
Out[76]:
0    a
1    b
2    (c)
3    d
```

(continues on next page)

(continued from previous page)

```
4    [e]
5    [stuff]
6    f
7    (g, h)
```

It is used everywhere where values are separated by ".".

You can see a complete example with a DotProxyList, like for the CommaProxyList, here: dotproxylist.

7.6.8 LineProxyList

LineProxyList is used to handle lines of code, it takes care to place the correct endl node between and to set the correct indentation and not to break the indentation of the next block (if there is one).

One particularity of LineProxyList is that it shows you explicitly the empty line (while other proxy lists never show you formatting). This is done because you'll often want to be able to manage those blank lines because you want to put some space in your code or separate group of lines.

```
In [77]: red = RedBaron("while 42:\n    stuff\n    other_stuff\n\n    there_is_an_
↪empty_line_before_me")

In [78]: red
Out[78]:
0   while 42:
        stuff
        other_stuff

        there_is_an_empty_line_before_me


In [79]: red[0].value
Out[79]:
0   stuff
1   other_stuff
2   '\n    '
3   there_is_an_empty_line_before_me

In [80]: red[0].append("plouf")

In [81]: red
Out[81]:
0   while 42:
        stuff
        other_stuff

        there_is_an_empty_line_before_me
        plouf


In [82]: red[0].value
Out[82]:
0   stuff
1   other_stuff
2   '\n    '
3   there_is_an_empty_line_before_me
4   plouf
```

You can see a complete example with a LineProxyList, like for the CommaProxyList, here: lineproxylist.

7.6.9 DecoratorLineProxyList

A DecoratorLineProxyList is exactly the same as a LineProxyList except it has a small modification to indent decorators correctly. Just think of it as a simple LineProxyList and everything will be fine.

Don't forget to add the :file:'@' when you add a new decorator (omitting it will raise an exception).

Example:

```
In [83]: red = RedBaron("@plop\ndef stuff():\n    pass\n")

In [84]: red
Out[84]:
0   @plop
    def stuff():
        pass


In [85]: red[0].decorators.append("@plouf")

In [86]: red[0].decorators
Out[86]:
0   @plop
1   @plouf

In [87]: red
Out[87]:
0   @plop
    @plouf
    def stuff():
        pass
```

Next

To learn about various helpers and features in RedBaron, read *Other*. Be sure to check the `.replace()` method on that page as it can be very useful.

7.7 Other

List of other features of RedBaron.

7.7.1 .parent

Every node and node list have a `.parent` attribute that points to the parent node or node list. If the node doesn't have a parent node (for example the node list returned when constructing a new instance using the `RedBaron` class), the `parent` attribute is set at `None`. A new node or node list created using `.copy()` always have its `parent` attribute set at `None`.

The attribute on which the node is assigned on the parent node is store in the `on_attribute` attribute. `on_attribute` is set at `"root"` if the parent is a RedBaron instance.

```
In [1]: red = RedBaron("a = 1 + caramba")

In [2]: red.help()
0 -----------------------------------------------------
AssignmentNode()
  # identifiers: assign, assignment, assignment_, assignmentnode
  operator=''
  target ->
    NameNode()
      # identifiers: name, name_, namenode
      value='a'
  annotation ->
    None
  value ->
    BinaryOperatorNode()
      # identifiers: binary_operator, binary_operator_, binaryoperator,␣
↪binaryoperatornode
      value='+'
      first ->
        IntNode() ...
      second ->
        NameNode() ...

In [3]: red.parent

In [4]: red.on_attribute

In [5]: red[0].parent
Out[5]: 0   a = 1 + caramba

In [6]: red[0].on_attribute
Out[6]: 'root'

In [7]: red[0].target.parent
Out[7]: a = 1 + caramba

In [8]: red[0].target.on_attribute
Out[8]: 'target'

In [9]: red[0].value.parent
Out[9]: a = 1 + caramba

In [10]: red[0].value.on_attribute
Out[10]: 'value'

In [11]: red[0].value.first.parent
Out[11]: 1 + caramba

In [12]: red[0].value.first.on_attribute
Out[12]: 'first'
```

7.7.2 .parent_find()

A helper method that allow you to do the equivalent of the `.find()` method but in the chain of the parents of the node. This is the equivalent of doing: `while node has a parent:  if node.parent match query: return node.parent, else:  node = node.parent`. It returns `None` if no parent match the query.

```
In [13]: red = RedBaron("def a():\n    with b:\n        def c():\n            pass")

In [14]: red.help()
0 -----------------------------------------------------
DefNode()
  # identifiers: def, def_, defnode, funcdef, funcdef_
  # default test value: name
  async=False
  name='a'
  return_annotation ->
    None
  decorators ->
  arguments ->
  value ->
    * WithNode()
        # identifiers: with, with_, withnode
        async=False
        contexts ->
          * WithContextItemNode() ...
        value ->
          * DefNode() ...

In [15]: r = red.pass_

In [16]: r
Out[16]: pass

In [17]: r.parent
Out[17]:
def c():
            pass

In [18]: r.parent_find('def')
Out[18]:
def c():
            pass

In [19]: r.parent_find('def', name='a')
Out[19]:
def a():
    with b:
        def c():
            pass

In [20]: r.parent_find('def', name='dont_exist')
```

7.7.3 .next .previous .next_recursive .previous_recursive .next_generator() .previous_generator()

In a similar fashion, nodes have a `.next` and `.previous` attributes that point to the next or previous node if the node is located in a node list. They are set at `None` if there is not adjacent node or if the node is not in a node list. A node list will never have a `.next` or `.previous` node, so those attributes will always be set at `None`.

Nodes also have a `.next_generator()` and `.previous_generator()` if you want to iterate on the neighbours of the node.

Nodes have also a `.next_recursive` and `.previous_recursive` attribute. It is similar to the non recursive

function but differ in the fact that, when using `.next_recursive` on a node at the end of the list, it points to the first adjacent node that exist in the parent hierarchy.

```
In [21]: red = RedBaron("[1, 2, 3]; a = 1")

In [22]: list = red[0]

In [23]: print(list.next)
;

In [24]: list.help()
ListNode()
  # identifiers: list, list_, listnode
  value ->
    * IntNode()
        # identifiers: int, int_, intnode
        value='1'
    * IntNode()
        # identifiers: int, int_, intnode
        value='2'
    * IntNode()
        # identifiers: int, int_, intnode
        value='3'

In [25]: assign = red[2]

In [26]: assign.help()
AssignmentNode()
  # identifiers: assign, assignment, assignment_, assignmentnode
  operator=''
  target ->
    NameNode()
      # identifiers: name, name_, namenode
      value='a'
  annotation ->
    None
  value ->
    IntNode()
      # identifiers: int, int_, intnode
      value='1'

In [27]: list.value[2].help(deep=1)
IntNode()
  # identifiers: int, int_, intnode
  value='3'
```

7.7.4 .next_intuitive/.previous_intuitive

Due to its tree nature, navigating in the FST might not behave as the user expect it. For example: doing a `.next` on a `TryNode` will not return the first `ExceptNode` (or `FinallyNode`) but will return the node after the try-excepts-else-finally node because it is a full node in itself in the FST.

See for yourself:

```
In [28]: red = RedBaron("try:\n    pass\nexcept:\n    pass\nafter")
```

(continues on next page)

(continued from previous page)

```
In [29]: red.try_
Out[29]:
try:
    pass
except:
    pass

In [30]: red.try_.next
Out[30]: after

In [31]: red.help()
0 -----------------------------------------------------
TryNode()
  # identifiers: try, try_, trynode
  else ->
    None
  finally ->
    None
  value ->
    * PassNode()
        # identifiers: pass, pass_, passnode
  excepts ->
    * ExceptNode()
        # identifiers: except, except_, exceptnode
        delimiter=''
        exception ->
          None
        target ->
          None
        value ->
          * PassNode() ...
1 -----------------------------------------------------
NameNode()
  # identifiers: name, name_, namenode
  value='after'
```

To solve this issue `.next_intuitive` and `.previous_intuitive` have been introduced:

```
In [32]: red
Out[32]:
0   try:
        pass
    except:
        pass

1   after

In [33]: red.try_.next_intuitive
Out[33]:
except:
    pass

In [34]: red.try_.next_intuitive.next_intuitive
Out[34]: after
```

This also applies to `IfNode`, `ElifNode`, `ElseNode`, `ForNode` and `WhileNode` (both of the last one can have an else statement). This also works coming from nodes outsides of those previous nodes.

For `IfNode`, `ElifNode` and `ElseNode` **inside** an `IfelseblockNode`:

```
In [35]: red = RedBaron("before\nif a:\n    pass\nelif b:\n    pass\nelse:\n    ␣
↪pass\nafter")

In [36]: red
Out[36]:
0   before
1   if a:
        pass
    elif b:
        pass
    else:
        pass

2   after

In [37]: red[1].help()
IfelseblockNode()
  # identifiers: ifelseblock, ifelseblock_, ifelseblocknode
  value ->
    * IfNode()
        # identifiers: if, if_, ifnode
        test ->
          NameNode() ...
        value ->
          * PassNode() ...
    * ElifNode()
        # identifiers: elif, elif_, elifnode
        test ->
          NameNode() ...
        value ->
          * PassNode() ...
    * ElseNode()
        # identifiers: else, else_, elsenode
        value ->
          * PassNode() ...

In [38]: red[1]
Out[38]:
if a:
    pass
elif b:
    pass
else:
    pass

In [39]: red.if_.next
Out[39]:
elif b:
    pass

In [40]: red.if_.next_intuitive
Out[40]:
elif b:
    pass

In [41]: red.if_.next_intuitive.next_intuitive
```

(continues on next page)

(continued from previous page)

```
Out[41]:
else:
    pass

In [42]: red.if_.next_intuitive.next_intuitive.next_intuitive
Out[42]: after

In [43]: red.if_.next_intuitive.next_intuitive.next_intuitive.next_intuitive
```

Warning: There is a subtlety: `IfelseblockNode` is **unaffected** by this behavior: you have to use `next_intuitive` or `previous_intuitive` on `IfNode`, `ElifNode` and `ElseNode` **inside** IfelseblockNode.

But, if you do a `next_intuitive` or `previous_intuitive` or a node around `IfelseblockNode` it will jump to the first or last node **inside** the `IfelseblockNode`.

See this example

```
In [44]: red = RedBaron("before\nif a:\n    pass\nelif b:\n    pass\nelse:\n    ␣
↪pass\nafter")

In [45]: red[1].ifelseblock.next_intuitive  # similar to .next
Out[45]: after

In [46]: red[1].ifelseblock.next.previous  # this is the IfelseblockNode
Out[46]:
if a:
    pass
elif b:
    pass
else:
    pass

In [47]: red[1].ifelseblock.next.previous_intuitive  # this is the ElseNode
Out[47]:
else:
    pass

In [48]: red[1].ifelseblock.previous.next  # this is the IfelseblockNode
Out[48]:
if a:
    pass
elif b:
    pass
else:
    pass

In [49]: red[1].ifelseblock.previous.next_intuitive  # this is the IfNode
Out[49]:
if a:
    pass
```

For `ForNode`:

```
In [50]: red = RedBaron("for a in b:\n    pass\nelse:\n    pass\nafter")

In [51]: red
Out[51]:
0   for a in b:
        pass
    else:
        pass

1   after

In [52]: red[0].help()
ForNode()
  # identifiers: for, for_, fornode
  async=False
  iterator ->
    NameNode()
      # identifiers: name, name_, namenode
      value='a'
  target ->
    NameNode()
      # identifiers: name, name_, namenode
      value='b'
  else ->
    ElseNode()
      # identifiers: else, else_, elsenode
      value ->
        * PassNode() ...
  value ->
    * PassNode()
        # identifiers: pass, pass_, passnode

In [53]: red.for_
Out[53]:
for a in b:
    pass
else:
    pass

In [54]: red.for_.next
Out[54]: after

In [55]: red.for_.next_intuitive
Out[55]:
else:
    pass

In [56]: red.for_.next_intuitive.next_intuitive
Out[56]: after
```

For `WhileNode`:

```
In [57]: red = RedBaron("while a:\n    pass\nelse:\n    pass\nafter")

In [58]: red
Out[58]:
0   while a:
```

(continues on next page)

(continued from previous page)

```
        pass
    else:
        pass

1   after

In [59]: red[0].help()
WhileNode()
  # identifiers: while, while_, whilenode
  test ->
    NameNode()
      # identifiers: name, name_, namenode
      value='a'
  else ->
    ElseNode()
      # identifiers: else, else_, elsenode
      value ->
        * PassNode() ...
  value ->
    * PassNode()
        # identifiers: pass, pass_, passnode

In [60]: red.while_
Out[60]:
while a:
    pass
else:
    pass

In [61]: red.while_.next
Out[61]: after

In [62]: red.while_.next_intuitive
Out[62]:
else:
    pass

In [63]: red.while_.next_intuitive.next_intuitive
Out[63]: after
```

7.7.5 .root

Every node have the `.root` attribute (property) that returns the root node in which this node is located:

```
In [64]: red = RedBaron("def a(): return 42")

In [65]: red.int_
Out[65]: 42

In [66]: assert red.int_.root is red
```

7.7.6 .index_on_parent

Every node have the `.index_on_parent` attribute (property) that returns the index at which this node is store in its parent node list. If the node isn't stored in a node list, it returns `None`. If the node is stored in a proxy list (*Proxy List*), it's the index in the proxy list that is returned. to get the unproxified index use *.index_on_parent_raw*.

```
In [67]: red = RedBaron("a = [1, 2, 3]")

In [68]: red[0].value.value
Out[68]:
0   1
1   2
2   3

In [69]: red[0].value.value[2]
Out[69]: 3

In [70]: red[0].value.value[2].index_on_parent
Out[70]: 2

In [71]: red[0].value
Out[71]: [1, 2, 3]

In [72]: red[0].value.index_on_parent
```

7.7.7 .index_on_parent_raw

Same as *.index_on_parent* except that it always return the unproxified whether the node is stored in a proxy list or not.

```
In [73]: red = RedBaron("a = [1, 2, 3]")

In [74]: red[0].value.value.node_list
Out[74]:
0   1
1   ,
2   2
3   ,
4   3

In [75]: red[0].value.value.node_list[2]
Out[75]: 2

In [76]: red[0].value.value.node_list[2].index_on_parent_raw
Out[76]: 2
```

7.7.8 .filtered()

Node list comes with a small helper function: `.filtered()` that returns a **tuple** containing the "significant" node (nodes that aren't comma node, dot node, space node or endl node).

```
In [77]: red = RedBaron("[1, 2, 3]")

In [78]: red[0].value
Out[78]:
```

(continues on next page)

(continued from previous page)

```
0   1
1   2
2   3

In [79]: red[0].value.filtered()
Out[79]: (1, 2, 3)
```

Note: the fact that it's a tuple that is returned will probably evolve in the future into a node list proxy or something like that, I just don't have the time to do something better right now.

7.7.9 .indentation

Every node has the property `.indentation` that will return the indentation level of the node:

```
In [80]: red = RedBaron("while a:\n    pass")

In [81]: red[0].indentation
Out[81]: ''

In [82]: red[0].test.indentation
Out[82]: ''

In [83]: red.pass_.indentation
Out[83]: '    '

In [84]: red = RedBaron("while a: pass")

In [85]: red.pass_.indentation
Out[85]: ''
```

7.7.10 .increase_indentation() and .decrease_indentation()

Those 2 methods allow you to change the indentation of a part of the tree. They expect the number of spaces to add or to remove as first argument.

```
In [86]: red = RedBaron("def a():\n    if plop:\n        pass")

In [87]: red
Out[87]:
0   def a():
        if plop:
            pass


In [88]: red[0].value.increase_indentation(15)

In [89]: red
Out[89]:
0   def a():
                   if plop:
                       pass
```

(continues on next page)

(continued from previous page)

```
In [90]: red[0].value.decrease_indentation(15)

In [91]: red
Out[91]:
0   def a():
        if plop:
            pass
```

7.7.11 .to_python()

> **Warning:** Since RedBaron calls ast.literal_eval it can only parse the python code parsed by the python version you are using.
>
> For example if you are using a python version inferior to 3.6, *to_python* will crash on *100_000* because it is only supported since python 3.6

This method safely evaluate the current selected nodes. It wraps ast.literal_eval, therefor, and for security reasons, it only works on a subset of python: numbers, strings, lists, dicts, tuples, boolean and `None`. Of course, using this method on a list/dict/tuple containing values that aren't listed here will raise a `ValueError`.

```
In [92]: RedBaron("42")[0].value  # string
Out[92]: '42'

In [93]: RedBaron("42")[0].to_python()  # python int
Out[93]: 42

In [94]: RedBaron("'a' 'b'")[0].dumps()
Out[94]: "'a' 'b'"

In [95]: RedBaron("'a' 'b'")[0].to_python()
Out[95]: 'ab'

In [96]: RedBaron("u'unicode string'")[0].to_python()
Out[96]: u'unicode string'

In [97]: RedBaron("[1, 2, 3]")[0].to_python()
Out[97]: [1, 2, 3]

In [98]: RedBaron("(1, 2, 3)")[0].to_python()
Out[98]: (1, 2, 3)

In [99]: RedBaron("{'foo': 'bar'}")[0].to_python()
Out[99]: {'foo': 'bar'}

In [100]: RedBaron("False")[0].to_python()
Out[100]: False

In [101]: RedBaron("True")[0].to_python()
Out[101]: True

In [102]: print(RedBaron("None")[0].to_python())
None
```

7.7.12 .path()

Every node has a `path()` method that will return a `Path` object to it. Every path object has a `.node` attribute that point to the node and a `.to_baron_path` that returns a Baron Path namedtuple.

```
In [103]: red = RedBaron("while a:\n    pass")

In [104]: red.pass_
Out[104]: pass

In [105]: path = red.pass_.path()

In [106]: path
Out[106]: <Path(PassNode(pass) @ [0, 'value', 1]) object at 140486230578512>

In [107]: path.node
Out[107]: pass

In [108]: path.to_baron_path()
Out[108]: [0, 'value', 1]
```

7.7.13 Path class

RedBaron provides a Path class that represent a path to a node.

class redbaron.**Path**(*node*)

Holds the path to a FST node

Path(node): path coming from the node's root Path.from_baron_path(node, path): path going down the node following the given path

Note that the second argument "path" is a baron path, i.e. list of keys that can be given for example by redbaron.Path(node).to_baron_path()

The second form is useful when converting a path given by baron to a redbaron node

7.7.14 .map .filter .apply

RedBaron nodes list have 3 helper methods `.map`, `.filter` and `.apply` quite similar to python builtins (except for apply). The main difference is that they return a node list instance instead of a python buildin list.

- `.map` takes a callable (like a lambda or a function) that receive a node as first argument, this callable is applied on every node of the node list and a node list containing the return of those applies will be returned.
- `.filter` works like `.map` but instead of returning a node list of the return of the callable, it returns a node list that contains the nodes for which the callable returned `True` (or something considered `True` in python)
- `.apply` works like `.map` but instead of returning the result of the callable, it returns to original node.

```
In [109]: red = RedBaron("[1, 2, 3]")

In [110]: red('int')
Out[110]:
0   1
1   2
2   3
```

(continues on next page)

(continued from previous page)

```
In [111]: red('int').map(lambda x: x.to_python() + 1)
Out[111]:
0   2
1   3
2   4

In [112]: red('int').filter(lambda x: x.to_python() % 2 == 0)
Out[112]: 0   2
```

```
In [113]: red = RedBaron("a()\nb()\nc(x=y)")

In [114]: red('call')
Out[114]:
0   ()
1   ()
2   (x=y)

# FIXME
# red('call').map(lambda x: x.append_value("answer=42"))
In [115]: red('call')
Out[115]:
0   ()
1   ()
2   (x=y)

In [116]: red = RedBaron("a()\nb()\nc(x=y)")

# FIXME
# red('call').apply(lambda x: x.append_value("answer=42"))
```

7.7.15 .replace()

`.replace()` is a method that allow to replace **in place** a node by another one. Like every operation of this nature, you can pass a string, a dict, a list of length one or a node instance.

```
In [117]: red = RedBaron("a()\nb()\nc(x=y)")

In [118]: red[2].replace("1 + 2")

In [119]: red
Out[119]:
0   a()
1   b()
2   1 + 2

In [120]: red[-1].replace("plop")

In [121]: red
Out[121]:
0   a()
1   b()
2   plop
```

7.7.16 .edit()

Helper method that allow to edit the code of the current **node** into an editor. The result is parsed and replace the code of the current node.

```
In [122]: red = RedBaron("def a(): return 42")

# should be used like this: (I can't execute this code here, obviously)
# red.return_.edit()
```

By default, the editor is taken from the variable `EDITOR` in the environment variables. If this variable is not present, nano is used. You can use a different editor this way: `node.edit(editor="vim")`.

7.7.17 .absolute_bounding_box

The absolute bounding box of a node represents its top-left and bottom-right position relative to the fst's root node. The position is given as a tuple `(line, column)` with **both starting at 1**.

```
In [123]: red = RedBaron("def a(): return 42")

In [124]: red.funcdef.value.absolute_bounding_box
Out[124]: BoundingBox (Position (1, 10), Position (2, 0))
```

You can also get the bounding box of "string" nodes like the left parenthesis in the example above by giving the attribute's name to the `get_absolute_bounding_box_of_attribute()` method:

```
In [125]: red.funcdef.get_absolute_bounding_box_of_attribute('(')
Out[125]: BoundingBox (Position (1, 6), Position (1, 6))
```

This is impossible to do without giving the attribute's name as an argument since the left parenthesis is not a redbaron Node.

7.7.18 .bounding_box

Every node has the `bounding_box` property which holds the top-left and bottom-right position of the node. Compared to the `absolute_bounding_box` property, it assumes the node is the root node so the top-left position is always `(1, 1)`.

```
In [126]: red = RedBaron("def a(): return 42")

In [127]: red.funcdef.value.absolute_bounding_box
Out[127]: BoundingBox (Position (1, 10), Position (2, 0))

In [128]: red.funcdef.value.bounding_box
Out[128]: BoundingBox (Position (1, 1), Position (2, 0))
```

7.7.19 .find_by_position()

You can find which node is located at a given line and column:

```
In [129]: red = RedBaron("def a(): return 42")

In [130]: red.find_by_position((1, 5))
```

(continues on next page)

(continued from previous page)

```
Out[130]: def a(): return 42

In [131]: red.find_by_position((1, 6)) # '(' is not a redbaron node
Out[131]: def a(): return 42
```

7.7.20 .at()

Returns first node at specific line

```
In [132]: red = RedBaron("def a():\n return 42")

In [133]: red.at(1) # Gives DefNode
Out[133]:
def a():
 return 42

In [134]: red.at(2) # Gives ReturnNode
Out[134]: return 42
```

7.7.21 Node.from_fst()

`Node.from_fst()` is a helper class method that takes an FST node and return a RedBaron node instance. Except if you need to go down at a low level or that RedBaron doesn't provide the helper you need, you shouldn't use it.

```
In [135]: from redbaron import Node

In [136]: Node.from_fst({"type": "name", "value": "a"})
Out[136]: a
```

`Node.from_fst()` takes 2 optional keywords arguments: `parent` and `on_attribute` that should respectively be RedBaron node instance (the parent node) and a string (the attribute of the parent node on which this node is stored). See *.parent* doc for a better understanding of those 2 parameters.

```
In [137]: red = RedBaron("[1,]")

In [138]: new_name = Node.from_fst({"type": "name", "value": "a"}, parent=red[0], on_
↪attribute="value")

In [139]: red[0].value.append(new_name)
```

7.7.22 NodeList.from_fst()

Similarly to `Node.from_fst()`, `NodeList.from_fst()` is a helper class method that takes an FST node **list** and return a RedBaron node **list** instance. Similarly, you probably don't need to go so low level.

```
In [140]: from redbaron import NodeList

In [141]: NodeList.from_fst([{"type": "name", "value": "a"}, {'first_formatting': [],
↪'type': 'comma', 'second_formatting': [{'type': 'space', 'value': ' '}]}, {"type":
↪"name", "value": "b"}])
Out[141]:
```

(continues on next page)

(continued from previous page)

```
0   a
1   ,
2   b
```

7.7.23 .insert_before .insert_after

One thing you often wants to do is to insert things just after or before the node you've just got via query. Those helpers are here for that:

```
In [142]: red = RedBaron("foo = 42\nprint('bar')\n")

In [143]: red
Out[143]:
0   foo = 42
1   print('bar')

In [144]: red.print_.insert_before("baz")

In [145]: red
Out[145]:
0   foo = 42
1   baz
2   print('bar')

In [146]: red.print_.insert_after("foobar")

In [147]: red
Out[147]:
0   foo = 42
1   baz
2   print('bar')
3   foobar
```

Additionally, you can give an optional argument `offset` to insert more than one line after or before:

```
In [148]: red = RedBaron("foo = 42\nprint('bar')\n")

In [149]: red
Out[149]:
0   foo = 42
1   print('bar')

In [150]: red.print_.insert_before("baz", offset=1)

In [151]: red
Out[151]:
0   baz
1   foo = 42
2   print('bar')

In [152]: red[0].insert_after("foobar", offset=1)

In [153]: red
Out[153]:
0   baz
```

(continues on next page)

(continued from previous page)

```
1   foo = 42
2   foobar
3   print('bar')
```

CHAPTER 8

Reference

This is the reference page for every node type encountered in RedBaron and their specificities.

8.1 Nodes References Page

8.2 TopClass

8.2.1 CodeBlockNode

CodeBlockNode is a type of node that has a body composed of indented code like the DefNode or the IfNode. Great care has been taken on the SetAttr of their value so you don't have to take care about reindenting and other formatting details.

Demonstration:

```
In [1]: red = RedBaron("def function():\n    pass\n")

In [2]: red
Out[2]:
0   def function():
        pass


In [3]: red[0].value = "stuff"  # first '\n' will be added, indentation will be set

In [4]: red
Out[4]:
0   def function():
        stuff


In [5]: red[0].value = "                    bad_indent"
```

(continues on next page)

(continued from previous page)

```
In [6]: red
Out[6]:
0   def function():
        bad_indent


In [7]: red[0].value = " some\n stuff"

In [8]: red
Out[8]:
0   def function():
        some
        stuff
```

Some for indented cases:

```
In [9]: red = RedBaron("class A:\n    def __init__():\n        pass\n\n    def␣
↪plop():\n        pass")

In [10]: red.def_.value = "not_indented"

In [11]: red
Out[11]:
0   class A:
        def __init__():
            not_indented

        def plop():
            pass


In [12]: red.def_.value = "\n                              badly_indented"

In [13]: red
Out[13]:
0   class A:
        def __init__():
            badly_indented

        def plop():
            pass


In [14]: red.def_.value = "some\nstuff\nfoo\nbar\n\npouet"

In [15]: red
Out[15]:
0   class A:
        def __init__():
            some
            stuff
            foo
            bar

            pouet
```

(continues on next page)

(continued from previous page)

```
        def plop():
            pass
```

8.3 Nodes

8.3.1 ArgumentGeneratorComprehensionNode

A node representing generator passed as an argument during a function call.

```
In [16]: RedBaron("a(x for y in z)")[0].value[1].value[0].help(deep=True)
ArgumentGeneratorComprehensionNode()
  # identifiers: argument_generator_comprehension, argument_generator_comprehension_,␣
↪argumentgeneratorcomprehension, argumentgeneratorcomprehensionnode
  result ->
    NameNode()
      # identifiers: name, name_, namenode
      value='x'
  generators ->
    * ComprehensionLoopNode()
        # identifiers: comprehension_loop, comprehension_loop_, comprehensionloop,␣
↪comprehensionloopnode
        iterator ->
          NameNode()
            # identifiers: name, name_, namenode
            value='y'
        target ->
          NameNode()
            # identifiers: name, name_, namenode
            value='z'
        ifs ->
```

SetAttr

```
In [17]: red = RedBaron("a(x for y in z)")

In [18]: red
Out[18]: 0   a(x for y in z)

In [19]: red[0].value[1].value[0].result = "pouet"

In [20]: red
Out[20]: 0   a(pouet for y in z)

In [21]: red[0].value[1].value[0].generators = "for artichaut in courgette"

In [22]: red
Out[22]: 0   a(pouet for artichaut in courgette)
```

8.3.2 AssertNode

A node representing the assert statement.

```
In [23]: RedBaron("assert test, message")[0].help(deep=True)
AssertNode()
  # identifiers: assert, assert_, assertnode
  value ->
    NameNode()
      # identifiers: name, name_, namenode
      value='test'
  message ->
    NameNode()
      # identifiers: name, name_, namenode
      value='message'
```

SetAttr

```
In [24]: red = RedBaron("assert some_test")

In [25]: red
Out[25]: 0   assert some_test

In [26]: red[0].value = "1 == caramba()"

In [27]: red
Out[27]: 0   assert 1 == caramba()

In [28]: red[0].message = "'foo bar'"

In [29]: red
Out[29]: 0   assert 1 == caramba(), 'foo bar'

In [30]: red[0].message = ""

In [31]: red
Out[31]: 0   assert 1 == caramba()
```

8.3.3 AssignmentNode

A node representing the assign operation in python (`foo = bar`) and the "augmented" assign (`foo += bar`).

```
In [32]: RedBaron("a = b")[0].help(deep=True)
AssignmentNode()
  # identifiers: assign, assignment, assignment_, assignmentnode
  operator=''
  target ->
    NameNode()
      # identifiers: name, name_, namenode
      value='a'
  annotation ->
    None
  value ->
    NameNode()
      # identifiers: name, name_, namenode
      value='b'

In [33]: RedBaron("a += b")[0].help(deep=True)
```

(continues on next page)

(continued from previous page)

```
AssignmentNode()
  # identifiers: assign, assignment, assignment_, assignmentnode
  operator='+'
  target ->
    NameNode()
      # identifiers: name, name_, namenode
      value='a'
  annotation ->
    None
  value ->
    NameNode()
      # identifiers: name, name_, namenode
      value='b'
```

SetAttr

Works as expected:

```
In [34]: red = RedBaron("a = b")

In [35]: red[0].first = "caramba"

In [36]: red
Out[36]: 0   a = b

In [37]: red[0].second = "42"

In [38]: red
Out[38]: 0   a = b
```

For the operator part, expected input should work:

```
In [39]: red = RedBaron("a = b")

In [40]: red[0].operator = "+="

In [41]: red
Out[41]: 0   a += b

In [42]: red[0].operator = "+" # equivalent to '+='

In [43]: red
Out[43]: 0   a += b

In [44]: red[0].operator = "-" # equivalent to '-='

In [45]: red
Out[45]: 0   a -= b

In [46]: red[0].operator = "=" # equivalent to '='

In [47]: red
Out[47]: 0   a = b

In [48]: red[0].operator = "/="
```

(continues on next page)

(continued from previous page)

```
In [49]: red
Out[49]: 0   a /= b

In [50]: red[0].operator = "" # equivalent to '='

In [51]: red
Out[51]: 0   a = b
```

8.3.4 AssociativeParenthesisNode

This node represents a statement prioritised on another by being surrounded by parenthesis. For e.g., the first part of this addition: `(1 + 1) * 2`.

```
In [52]: RedBaron("(foo)")[0].help(deep=True)
AssociativeParenthesisNode()
  # identifiers: associative_parenthesis, associative_parenthesis_,␣
↪associativeparenthesis, associativeparenthesisnode
  value ->
    NameNode()
      # identifiers: name, name_, namenode
      value='foo'
```

SetAttr

```
In [53]: red = RedBaron("(foo)")

In [54]: red
Out[54]: 0   (foo)

In [55]: red[0].value = "1 + 1"

In [56]: red
Out[56]: 0   (1 + 1)
```

8.3.5 AtomtrailersNode

This node represents a combination of NameNode, *DotNode*, *CallNode*, *GetitemNode* sorted in a list. For e.g.: `a.b().c[d]`.

```
In [57]: RedBaron("a.b().c[d]")[0].help(deep=True)
AtomtrailersNode()
  # identifiers: atomtrailers, atomtrailers_, atomtrailersnode
  value ->
    * NameNode()
        # identifiers: name, name_, namenode
        value='a'
    * NameNode()
        # identifiers: name, name_, namenode
        value='b'
    * CallNode()
```

(continues on next page)

(continued from previous page)

```
        # identifiers: call, call_, callnode
        value ->
    * NameNode()
        # identifiers: name, name_, namenode
        value='c'
    * GetitemNode()
        # identifiers: getitem, getitem_, getitemnode
        value ->
          NameNode()
            # identifiers: name, name_, namenode
            value='d'
```

SetAttr

```
In [58]: red = RedBaron("a.b()")

In [59]: red
Out[59]: 0   a.b()

In [60]: red[0].value = "d.be"

In [61]: red
Out[61]: 0   d.be
```

8.3.6 BinaryNode

The node represents a binary number value.

```
In [62]: RedBaron("0b10101")[0].help(deep=True)
BinaryNode()
  # identifiers: binary, binary_, binarynode
  value='0b10101'
```

8.3.7 BinaryOperatorNode

The node represents a binary operator (an operator (e.g.: + - /..) applied to 2 values) with its operands. For e.g.: `1 + 1`.

```
In [63]: RedBaron("1 + 1")[0].help(deep=True)
BinaryOperatorNode()
  # identifiers: binary_operator, binary_operator_, binaryoperator, binaryoperatornode
  value='+'
  first ->
    IntNode()
      # identifiers: int, int_, intnode
      value='1'
  second ->
    IntNode()
      # identifiers: int, int_, intnode
      value='1'
```

SetAttr

```
In [64]: red = RedBaron("1 + 1")

In [65]: red
Out[65]: 0   1 + 1

In [66]: red[0].value = "*"

In [67]: red
Out[67]: 0   1 * 1

In [68]: red[0].first = "(1 + 1)"

In [69]: red
Out[69]: 0   (1 + 1) * 1

In [70]: red[0].second = "caramba"

In [71]: red
Out[71]: 0   (1 + 1) * caramba
```

8.3.8 BooleanOperatorNode

The node represents a boolean operator (an operator (e.g.: `and` `or`) applied to 2 values) with its operands. For e.g.: `x and y`.

```
In [72]: RedBaron("x and y")[0].help(deep=True)
BooleanOperatorNode()
  # identifiers: boolean_operator, boolean_operator_, booleanoperator,␣
↪booleanoperatornode
  value='and'
  first ->
    NameNode()
      # identifiers: name, name_, namenode
      value='x'
  second ->
    NameNode()
      # identifiers: name, name_, namenode
      value='y'
```

SetAttr

```
In [73]: red = RedBaron("x and y")

In [74]: red
Out[74]: 0   x and y

In [75]: red[0].value = "or"

In [76]: red
Out[76]: 0   x or y

In [77]: red[0].first = "plop"
```

(continues on next page)

(continued from previous page)

```
In [78]: red
Out[78]: 0   plop or y

In [79]: red[0].second = "oupsi"

In [80]: red
Out[80]: 0   plop or oupsi
```

8.3.9 CallNode

A node representing a call (eg: `a()`, here `a` is called with no arguments). It is always stored in an *AtomtrailersNode* or a *DecoratorNode*.

```
In [81]: RedBaron("a(b, c=d)")[0].value[1].help(deep=True)
CallNode()
  # identifiers: call, call_, callnode
  value ->
    * CallArgumentNode()
        # identifiers: call_argument, call_argument_, callargument, callargumentnode
        target ->
          None
        value ->
          NameNode()
            # identifiers: name, name_, namenode
            value='b'
    * CallArgumentNode()
        # identifiers: call_argument, call_argument_, callargument, callargumentnode
        target ->
          NameNode()
            # identifiers: name, name_, namenode
            value='c'
        value ->
          NameNode()
            # identifiers: name, name_, namenode
            value='d'
```

SetAttr

SetAttr works as expected:

```
In [82]: red = RedBaron("a()")

In [83]: red[0].value[1].value = "b, c=d, *e, **f"

In [84]: red
Out[84]: 0   a(b, c=d, *e, **f)
```

8.3.10 CallArgumentNode

A node representing an argument or a named argument of a *CallNode* (other nodes that can be in a CallNode are *ListArgumentNode* and *DictArgumentNode*).

```
In [85]: RedBaron("a(b, c=d)")[0].value[1].value[0].help(deep=True)
CallArgumentNode()
  # identifiers: call_argument, call_argument_, callargument, callargumentnode
  target ->
    None
  value ->
    NameNode()
      # identifiers: name, name_, namenode
      value='b'

In [86]: RedBaron("a(b, c=d)")[0].value[1].value[1].help(deep=True)
CallArgumentNode()
  # identifiers: call_argument, call_argument_, callargument, callargumentnode
  target ->
    NameNode()
      # identifiers: name, name_, namenode
      value='c'
  value ->
    NameNode()
      # identifiers: name, name_, namenode
      value='d'
```

SetAttr

```
In [87]: red = RedBaron("a(b)")

In [88]: red
Out[88]: 0   a(b)

In [89]: red[0].value[1].value[0] = "stuff=foo"

In [90]: red
Out[90]: 0   a(stuff=foo)
```

8.3.11 ClassNode

A node representing a class definition.

```
In [91]: RedBaron("class SomeAwesomeName(A, B, C): pass")[0].help(deep=True)
ClassNode()
  # identifiers: class, class_, classnode
  # default test value: name
  name='SomeAwesomeName'
  parenthesis=True
  decorators ->
  inherit_from ->
    * TupleNode()
        # identifiers: tuple, tuple_, tuplenode
        with_parenthesis=False
        value ->
          * NameNode()
              # identifiers: name, name_, namenode
              value='A'
          * NameNode()
```

(continues on next page)

(continued from previous page)

```
            # identifiers: name, name_, namenode
            value='B'
        * NameNode()
            # identifiers: name, name_, namenode
            value='C'
  value ->
    * PassNode()
        # identifiers: pass, pass_, passnode
```

SetAttr

ClassNode is a CodeBlockNode which means its value attribute accepts a wide range of values, see *CodeBlockNode* for more information. Most other attributes work as expected:

```
In [92]: red = RedBaron("class SomeAwesomeName(A, B, C): pass")

In [93]: red[0].name = "AnotherAwesomeName"

In [94]: red
Out[94]:
0   class AnotherAwesomeName(A, B, C): pass


In [95]: red[0].inherit_from = "object"

In [96]: red
Out[96]:
0   class AnotherAwesomeName(object): pass
```

8.3.12 CommaNode

A node representing a comma, this is the kind of formatting node that you might have to deal with if not enough high level helpers are available. They are generally present in call, function arguments definition and data structure sugar syntactic notation.

The comma node is responsible for holding the formatting around it.

```
In [97]: RedBaron("[1, 2, 3]")[0].value.node_list[1].help(deep=True)
CommaNode()
  # identifiers: comma, comma_, commanode
```

8.3.13 ComparisonNode

The node represents a comparison operation, for e.g.: `42 > 30`.

```
In [98]: RedBaron("42 > 30")[0].help(deep=True)
ComparisonNode()
  # identifiers: comparison, comparison_, comparisonnode
  first ->
    IntNode()
      # identifiers: int, int_, intnode
      value='42'
```

(continues on next page)

(continued from previous page)

```
  value ->
    ComparisonOperatorNode()
      # identifiers: comparison_operator, comparison_operator_, comparisonoperator,␣
↪comparisonoperatornode
      first='>'
      second=''
  second ->
    IntNode()
      # identifiers: int, int_, intnode
      value='30'
```

SetAttr

```
In [99]: red = RedBaron("42 > 30")

In [100]: red
Out[100]: 0   42 > 30

In [101]: red[0].operator = "=="

In [102]: red
Out[102]: 0   42 > 30

In [103]: red[0].first = "(1 + 1)"

In [104]: red
Out[104]: 0   (1 + 1) > 30

In [105]: red[0].second = "caramba"

In [106]: red
Out[106]: 0   (1 + 1) > caramba
```

8.3.14 ComprehensionIfNode

The node represents "if" condition in a comprehension loop. It is always a member of a *ComprehensionLoopNode*.

```
In [107]: RedBaron("[x for x in x if condition]")[0].generators[0].ifs[0].
↪help(deep=True)
ComprehensionIfNode()
  # identifiers: comprehension_if, comprehension_if_, comprehensionif,␣
↪comprehensionifnode
  value ->
    NameNode()
      # identifiers: name, name_, namenode
      value='condition'
```

SetAttr

```
In [108]: red = RedBaron("[x for x in x if condition]")
```

(continues on next page)

(continued from previous page)

```
In [109]: red
Out[109]: 0   [x for x in x if condition]

In [110]: red[0].generators[0].ifs[0].value = "True"

In [111]: red
Out[111]: 0   [x for x in x if True]
```

8.3.15 ComprehensionLoopNode

The node represents the loop part of a comprehension structure.

```
In [112]: RedBaron("[x for y in z]")[0].generators[0].help(deep=True)
ComprehensionLoopNode()
  # identifiers: comprehension_loop, comprehension_loop_, comprehensionloop,␣
↪comprehensionloopnode
  iterator ->
    NameNode()
      # identifiers: name, name_, namenode
      value='y'
  target ->
    NameNode()
      # identifiers: name, name_, namenode
      value='z'
  ifs ->
```

SetAttr

```
In [113]: red = RedBaron("[x for y in z]")

In [114]: red
Out[114]: 0   [x for y in z]

In [115]: red[0].generators[0].target = "plop"

In [116]: red
Out[116]: 0   [x for y in plop]

In [117]: red[0].generators[0].iterator = "iter"

In [118]: red
Out[118]: 0   [x for iter in plop]

In [119]: red[0].generators[0].ifs = "if a if b"

In [120]: red
Out[120]: 0   [x for iter in plop if a if b]
```

8.3.16 DecoratorNode

A node representing an individual decorator (of a function or a class).

```
In [121]: RedBaron("@stuff.plop(*a)\ndef b(): pass")[0].decorators[0].help(deep=True)
DecoratorNode()
  # identifiers: decorator, decorator_, decoratornode
  value ->
    DottedNameNode()
      # identifiers: dotted_name, dotted_name_, dottedname, dottednamenode
      value ->
        * NameNode()
            # identifiers: name, name_, namenode
            value='stuff'
        * DotNode()
            # identifiers: dot, dot_, dotnode
        * NameNode()
            # identifiers: name, name_, namenode
            value='plop'
  call ->
    CallNode()
      # identifiers: call, call_, callnode
      value ->
        * ListArgumentNode()
            # identifiers: list_argument, list_argument_, listargument,␣
↪listargumentnode
            value ->
              NameNode()
                # identifiers: name, name_, namenode
                value='a'
            annotation ->
              None
```

SetAttr

```
In [122]: red = RedBaron("@stuff\ndef a(): pass")

In [123]: red
Out[123]:
0   @stuff
    def a(): pass


In [124]: red[0].decorators[0].value = "a.b.c"

In [125]: red
Out[125]:
0   @a.b.c
    def a(): pass


In [126]: red[0].decorators[0].call = "(*args)"

In [127]: red
Out[127]:
0   @a.b.c(*args)
    def a(): pass


In [128]: red[0].decorators[0].call = ""
```

(continues on next page)

(continued from previous page)

```
In [129]: red
Out[129]:
0   @a.b.c
    def a(): pass
```

8.3.17 DefNode

A node representing a function definition.

```
In [130]: RedBaron("def stuff():\n    pass\n")[0].help(deep=True)
DefNode()
  # identifiers: def, def_, defnode, funcdef, funcdef_
  # default test value: name
  async=False
  name='stuff'
  return_annotation ->
    None
  decorators ->
  arguments ->
  value ->
    * PassNode()
        # identifiers: pass, pass_, passnode
```

SetAttr

DefNode is a CodeBlockNode which means its value attribute accepts a wide range of values, see *CodeBlockNode* for more information. Most other attributes works as expected:

```
In [131]: red = RedBaron("def stuff():\n    body\n")

In [132]: red[0]
Out[132]:
def stuff():
    body

In [133]: red[0].name = "awesome_function"

In [134]: red[0].arguments = "a, b=None, *c, **d"

In [135]: red
Out[135]:
0   def awesome_function(a, b=None, *c, **d):
        body
```

Decorators might be a bit less intuitive:

```
In [136]: red =  RedBaron("def stuff():\n    body\n")

In [137]: red[0].decorators = "@foo(*plop)"

In [138]: red
Out[138]:
```

(continues on next page)

(continued from previous page)

```
0   @foo(*plop)
    def stuff():
        body


In [139]: red[0].decorators = "@foo\n@bar.baz()"

In [140]: red
Out[140]:
0   @foo
    @bar.baz()
    def stuff():
        body


In [141]: red[0].decorators = "    @pouet"  # SetAttr will take care of reindenting␣
↪everything as expected

In [142]: red
Out[142]:
0   @pouet
    def stuff():
        body
```

New in 0.7.

Async is a boolean attribute that determine if a function is async:

```
In [143]: red =  RedBaron("def stuff():\n    body\n")

In [144]: red[0].async_
Out[144]: False

In [145]: red[0].async_ = True

In [146]: red
Out[146]:
0   async def stuff():
        body


In [147]: red[0].async_ = False

In [148]: red
Out[148]:
0   def stuff():
        body
```

Warning: As of python 3.7 *async* and *await* are now reserved keywords so don't uses *red.async*, it works as expected but won't make your code forward compatible.

New in 0.9

Return annotation management:

```
In [149]: red =  RedBaron("def stuff():\n    return 42\n")

In [150]: red
Out[150]:
0   def stuff():
        return 42


In [151]: red[0].return_annotation = "Int"

In [152]: red
Out[152]:
0   def stuff() -> Int:
        return 42


In [153]: red[0].return_annotation = ""

In [154]: red
Out[154]:
0   def stuff():
        return 42
```

8.3.18 DefArgumentNode

A node representing an argument in a function definition.

```
In [155]: RedBaron("def a(b, c=d): pass")[0].arguments.help(deep=True)
0 -----------------------------------------------------
DefArgumentNode()
  # identifiers: def_argument, def_argument_, defargument, defargumentnode
  target ->
    NameNode()
      # identifiers: name, name_, namenode
      value='b'
  annotation ->
    None
  value ->
    None
1 -----------------------------------------------------
CommaNode()
  # identifiers: comma, comma_, commanode
2 -----------------------------------------------------
DefArgumentNode()
  # identifiers: def_argument, def_argument_, defargument, defargumentnode
  target ->
    NameNode()
      # identifiers: name, name_, namenode
      value='c'
  annotation ->
    None
  value ->
    NameNode()
      # identifiers: name, name_, namenode
      value='d'
```

SetAttr

```
In [156]: red = RedBaron("def a(b): pass")

In [157]: red
Out[157]:
0   def a(b): pass


In [158]: red[0].arguments[0].name = "plop"

In [159]: red
Out[159]:
0   def a(b): pass


In [160]: red[0].arguments[0].value = "1 + 1"

In [161]: red
Out[161]:
0   def a(b=1 + 1): pass
```

New in 0.9

Annotations:

```
In [162]: red = RedBaron("def a(b): pass")

In [163]: red
Out[163]:
0   def a(b): pass


In [164]: red[0].arguments[0].annotation = "Int"

In [165]: red
Out[165]:
0   def a(b : Int): pass


In [166]: red[0].arguments[0].annotation
Out[166]: Int

In [167]: red
Out[167]:
0   def a(b : Int): pass
```

8.3.19 DelNode

A node representing a `del` statement.

```
In [168]: RedBaron("del stuff")[0].help(deep=True)
DelNode()
  # identifiers: del, del_, delnode
  value ->
    NameNode()
```

(continues on next page)

(continued from previous page)

```
    # identifiers: name, name_, namenode
    value='stuff'
```

SetAttr

```
In [169]: red = RedBaron("del stuff")

In [170]: red
Out[170]: 0   del stuff

In [171]: red[0].value = "some, other, stuff"

In [172]: red
Out[172]: 0   del some, other, stuff
```

8.3.20 DictArgumentNode

A node representing a 'kwargs' defined in a function definition argument or used in a *CallNode*.

```
In [173]: RedBaron("a(**b)")[0].value[1].value[0].help(deep=True)
DictArgumentNode()
  # identifiers: dict_argument, dict_argument_, dictargument, dictargumentnode
  value ->
    NameNode()
      # identifiers: name, name_, namenode
      value='b'
  annotation ->
    None
```

SetAttr

```
In [174]: red = RedBaron("a(**b)")

In [175]: red
Out[175]: 0   a(**b)

In [176]: red[0].value[1].value[0].value = "plop"

In [177]: red
Out[177]: 0   a(**plop)
```

New in 0.9

Annotations:

```
In [178]: red = RedBaron("def a(**b): pass")

In [179]: red
Out[179]:
0   def a(**b): pass
```

(continues on next page)

(continued from previous page)

```
In [180]: red[0].arguments[0].annotation = "Int"

In [181]: red
Out[181]:
0   def a(**b : Int): pass


In [182]: red[0].arguments[0].annotation
Out[182]: Int

In [183]: red
Out[183]:
0   def a(**b : Int): pass
```

8.3.21 DictNode

A node representing python sugar syntactic notation for dict.

```
In [184]: RedBaron("{'a': 1, 'b': 2, 'c': 3}")[0].help(deep=True)
DictNode()
  # identifiers: dict, dict_, dictnode
  value ->
    * DictitemNode()
        # identifiers: dictitem, dictitem_, dictitemnode
        key ->
          StringNode()
            # identifiers: string, string_, stringnode
            value="'a'"
        value ->
          IntNode()
            # identifiers: int, int_, intnode
            value='1'
    * DictitemNode()
        # identifiers: dictitem, dictitem_, dictitemnode
        key ->
          StringNode()
            # identifiers: string, string_, stringnode
            value="'b'"
        value ->
          IntNode()
            # identifiers: int, int_, intnode
            value='2'
    * DictitemNode()
        # identifiers: dictitem, dictitem_, dictitemnode
        key ->
          StringNode()
            # identifiers: string, string_, stringnode
            value="'c'"
        value ->
          IntNode()
            # identifiers: int, int_, intnode
            value='3'
```

8.3.22 DictComprehensionNode

A node representing dictionary comprehension node.

```
In [185]: RedBaron("{a: b for c in d}")[0].help(deep=True)
DictComprehensionNode()
  # identifiers: dict_comprehension, dict_comprehension_, dictcomprehension,␣
↪dictcomprehensionnode
  result ->
    DictitemNode()
      # identifiers: dictitem, dictitem_, dictitemnode
      key ->
        NameNode()
          # identifiers: name, name_, namenode
          value='a'
      value ->
        NameNode()
          # identifiers: name, name_, namenode
          value='b'
  generators ->
    * ComprehensionLoopNode()
        # identifiers: comprehension_loop, comprehension_loop_, comprehensionloop,␣
↪comprehensionloopnode
        iterator ->
          NameNode()
            # identifiers: name, name_, namenode
            value='c'
        target ->
          NameNode()
            # identifiers: name, name_, namenode
            value='d'
        ifs ->
```

SetAttr

```
In [186]: red = RedBaron("{a: b for c in d}")

In [187]: red
Out[187]: 0   {a: b for c in d}

In [188]: red[0].result = "plop: poulpe"

In [189]: red
Out[189]: 0   {plop: poulpe for c in d}

In [190]: red[0].generators = "for zomg in wtf"

In [191]: red
Out[191]: 0   {plop: poulpe for zomg in wtf}
```

8.3.23 DottedAsNameNode

A node representing an argument to the import node.

```
In [192]: RedBaron("import a.b.c as d")[0].value[0].help(deep=True)
DottedAsNameNode()
  # identifiers: dotted_as_name, dotted_as_name_, dottedasname, dottedasnamenode
  target='d'
  value ->
    * NameNode()
        # identifiers: name, name_, namenode
        value='a'
    * NameNode()
        # identifiers: name, name_, namenode
        value='b'
    * NameNode()
        # identifiers: name, name_, namenode
        value='c'
```

SetAttr

```
In [193]: red = RedBaron("import a.b.c as d")

In [194]: red
Out[194]: 0   import a.b.c as d

In [195]: red[0].value[0].value = "some.random.module"

In [196]: red
Out[196]: 0   import some.random.module as d

In [197]: red[0].value[0].target = "stuff"

In [198]: red
Out[198]: 0   import some.random.module as stuff
```

8.3.24 DotNode

A node representing a dot '.', generally found in atom trailers (this kind of structure: 'variable.another_variable(call)[getitem]'). This is the kind of formatting node that you might have to deal with if not enough high level helpers are available.

The dot node is responsible for holding the formatting around it.

```
In [199]: RedBaron("a.b")[0].value[1].help(deep=True)
NameNode()
  # identifiers: name, name_, namenode
  value='b'
```

8.3.25 ElifNode

A node representing an elif statement.

The ElifNode, like the IfNode or the *ElseNode* are stored in a *IfelseblockNode*.

```
In [200]: RedBaron("if a: pass\nelif b: pass")[0].value[1].help(deep=True)
ElifNode()
  # identifiers: elif, elif_, elifnode
  test ->
    NameNode()
      # identifiers: name, name_, namenode
      value='b'
  value ->
    * PassNode()
        # identifiers: pass, pass_, passnode
```

SetAttr

ElifNode is a CodeBlockNode which means its value attribute accepts a wide range of values, see *CodeBlockNode* for more information. Other attributes work as expected:

```
In [201]: red = RedBaron("if a: pass\nelif b: pass")

In [202]: red
Out[202]:
0   if a: pass
    elif b: pass


In [203]: red[0].value[1].test = "1 + 1 == 11"

In [204]: red
Out[204]:
0   if a: pass
    elif 1 + 1 == 11: pass
```

8.3.26 ElseNode

A node representing an else statement.

The ElseNode, like the IfNode or the *ElifNode*, is stored in a *IfelseblockNode*.

```
In [205]: RedBaron("if a: pass\nelse: pass")[0].value[1].help(deep=True)
ElseNode()
  # identifiers: else, else_, elsenode
  value ->
    * PassNode()
        # identifiers: pass, pass_, passnode
```

SetAttr

ElifNode is a CodeBlockNode which means its value attribute accepts a wide range of values, see *CodeBlockNode* for more information.

8.3.27 EllipsisNode

A node representing "…".

```
In [206]: RedBaron("def a(): ...").ellipsis.help(deep=True)
EllipsisNode()
  # identifiers: ellipsis, ellipsis_, ellipsisnode
```

8.3.28 EndlNode

A node for the end line ('n', 'rn') component.

This node is responsible for holding the indentation AFTER itself. This node also handles formatting around it, CommentNode **before** an EndlNode will end up in the formatting key of an EndlNode 99% of the time (the exception is if the CommentNode is the last node of the file).

```
In [207]: RedBaron("\n")[0].help()
EndlNode()
  # identifiers: endl, endl_, endlnode
  value='\n'
  indent=''

In [208]: RedBaron("# first node of the file\n# last node of the file").node_list.
↪help()
0 -----------------------------------------------------
CommentNode()
  # identifiers: comment, comment_, commentnode
  value='# first node of the file'
1 -----------------------------------------------------
EndlNode()
  # identifiers: endl, endl_, endlnode
  value='\n'
  indent=''
2 -----------------------------------------------------
CommentNode()
  # identifiers: comment, comment_, commentnode
  value='# last node of the file'
```

8.3.29 ExceptNode

A node representing an except statement (member of a *TryNode*).

```
In [209]: RedBaron("try: pass\nexcept FooBar: pass\nexcept Exception: pass\nelse:␣
↪pass\nfinally: pass\n")[0].excepts[0].help(deep=True)
ExceptNode()
  # identifiers: except, except_, exceptnode
  delimiter=''
  exception ->
    NameNode()
      # identifiers: name, name_, namenode
      value='FooBar'
  target ->
    None
  value ->
    * PassNode()
        # identifiers: pass, pass_, passnode
```

SetAttr

ExceptNode is a CodeBlockNode which means its value attribute accepts a wide range of values, see *CodeBlockNode* for more information. Other attributes work as expected:

```
In [210]: red = RedBaron("try: pass\nexcept: pass")

In [211]: red
Out[211]:
0   try: pass
    except: pass


In [212]: red[0].excepts[0].exception = "plop"

In [213]: red
Out[213]:
0   try: pass
    except plop: pass


In [214]: red[0].excepts[0].target = "stuff"

In [215]: red
Out[215]:
0   try: pass
    except plop as stuff: pass


In [216]: red[0].excepts[0].exception = ""

In [217]: red
Out[217]:
0   try: pass
    except: pass


# red[0].excepts[0].target = "stuff" <- would raise without a target
```

8.3.30 ExecNode

A node representing an exec statement.

```
In [218]: RedBaron("exec '1 + 1' in a, b")[0].help(deep=True)
ExecNode()
  # identifiers: exec, exec_, execnode
  value ->
    StringNode()
      # identifiers: string, string_, stringnode
      value="'1 + 1'"
  globals ->
    NameNode()
      # identifiers: name, name_, namenode
      value='a'
  locals ->
    NameNode()
```

(continues on next page)

(continued from previous page)

```
      # identifiers: name, name_, namenode
      value='b'
```

SetAttr

```
In [219]: red = RedBaron("exec 'stuff'")

In [220]: red
Out[220]: 0   exec 'stuff'

In [221]: red[0].value = 'some_code'

In [222]: red
Out[222]: 0   exec some_code

In [223]: red[0].globals = 'x'

In [224]: red
Out[224]: 0   exec some_code in x

In [225]: red[0].locals = 'y'

In [226]: red
Out[226]: 0   exec some_code in x, y
```

8.3.31 FinallyNode

A node representing a finally statement (member of a *TryNode*).

```
In [227]: RedBaron("try: pass\nexcept FooBar: pass\nexcept Exception: pass\nelse:␣
↪pass\nfinally: pass\n").finally_.help(deep=True)
FinallyNode()
  # identifiers: finally, finally_, finallynode
  value ->
    * PassNode()
        # identifiers: pass, pass_, passnode
```

SetAttr

FinallyNode is a CodeBlockNode which means its value attribute accepts a wide range of values, see *CodeBlockNode* for more information.

8.3.32 ForNode

A node representing a for loop.

```
In [228]: RedBaron("for i in b:\n    pass")[0].help(deep=True)
ForNode()
  # identifiers: for, for_, fornode
  async=False
```

(continues on next page)

(continued from previous page)

```
  iterator ->
    NameNode()
      # identifiers: name, name_, namenode
      value='i'
  target ->
    NameNode()
      # identifiers: name, name_, namenode
      value='b'
  else ->
    None
  value ->
    * PassNode()
        # identifiers: pass, pass_, passnode
```

SetAttr

ForNode is a CodeBlockNode which means its value attribute accepts a wide range of values, see *CodeBlockNode* for more information. The else attributes accept a great ranges of inputs, since `else` is a reserved python keyword, you need to access it using the `else_` attribute. Other attributes work as expected:

```
In [229]: red = RedBaron("for i in b: pass")

In [230]: red
Out[230]:
0   for i in b: pass


In [231]: red[0].iterator = "i, j, k"

In [232]: red
Out[232]:
0   for i, j, k in b: pass


In [233]: red[0].target = "[x for x in stuff if condition]"

In [234]: red
Out[234]:
0   for i, j, k in [x for x in stuff if condition]: pass


In [235]: red[0].else_ = "do_stuff"

In [236]: red
Out[236]:
0   for i, j, k in [x for x in stuff if condition]: pass
    else:
        do_stuff


In [237]: red[0].else_ = "else: foobar"

In [238]: red
Out[238]:
0   for i, j, k in [x for x in stuff if condition]: pass
```

(continues on next page)

(continued from previous page)

```
    else:
        foobar


In [239]: red[0].else_ = "    else:\n        badly_indented_and_trailing\n\n\n\n"

In [240]: red
Out[240]:
0   for i, j, k in [x for x in stuff if condition]: pass
    else:
        badly_indented_and_trailing
```

New in 0.8.

Async is a boolean attribute that determine if a function is async:

```
In [241]: red =  RedBaron("for a in b: pass")

In [242]: red[0].async_
Out[242]: False

In [243]: red[0].async_ = True

In [244]: red
Out[244]:
0   async for a in b: pass


In [245]: red[0].async_ = False

In [246]: red
Out[246]:
0   for a in b: pass
```

Warning: As of python 3.7 *async* and *await* are now reserved keywords so don't uses *red.async*, it works as expected but won't make your code forward compatible.

8.3.33 FromImportNode

A node representing a "from import" statement.

```
In [247]: RedBaron("from a import b")[0].help(deep=True)
FromImportNode()
  # identifiers: from_import, from_import_, fromimport, fromimportnode
  # helpers: full_path_modules, full_path_names, modules, names
  value ->
    * NameNode()
        # identifiers: name, name_, namenode
        value='a'
  targets ->
    * NameAsNameNode()
        # identifiers: name_as_name, name_as_name_, nameasname, nameasnamenode
        value='b'
        target=''
```

SetAttr

```
In [248]: red = RedBaron("from a import b")

In [249]: red
Out[249]: 0   from a import b

In [250]: red[0].value = "some.module"

In [251]: red
Out[251]: 0   from some.module import b

In [252]: red[0].targets = "a as b, c as d, e"

In [253]: red
Out[253]: 0   from some.module import a as b, c as d, e
```

Helpers

To reduce the complexity, 2 helpers method are provided:

```
In [254]: red = RedBaron("from foo.bar import baz as stuff, plop")

In [255]: red[0].names()   # names added to the context
Out[255]: ['stuff', 'plop']

In [256]: red[0].modules()   # modules imported
Out[256]: ['baz', 'plop']

In [257]: red[0].full_path_names()   # names added to the context with full path
Out[257]: ['foo.bar.stuff', 'foo.bar.plop']

In [258]: red[0].full_path_modules()   # modules imported with full path
Out[258]: ['foo.bar.baz', 'foo.bar.plop']
```

8.3.34 GeneratorComprehensionNode

A node representing a generator comprehension node.

```
In [259]: RedBaron("(x for y in z)")[0].help(deep=True)
GeneratorComprehensionNode()
  # identifiers: generator_comprehension, generator_comprehension_,␣
↪generatorcomprehension, generatorcomprehensionnode
  result ->
    NameNode()
      # identifiers: name, name_, namenode
      value='x'
  generators ->
    * ComprehensionLoopNode()
        # identifiers: comprehension_loop, comprehension_loop_, comprehensionloop,␣
↪comprehensionloopnode
        iterator ->
          NameNode()
            # identifiers: name, name_, namenode
```

(continues on next page)

(continued from previous page)

```
          value='y'
      target ->
        NameNode()
          # identifiers: name, name_, namenode
          value='z'
      ifs ->
```

SetAttr

```
In [260]: red = RedBaron("(x for y in z)")

In [261]: red
Out[261]: 0   (x for y in z)

In [262]: red[0].result = "pouet"

In [263]: red
Out[263]: 0   (pouet for y in z)

In [264]: red[0].generators = "for artichaut in courgette"

In [265]: red
Out[265]: 0   (pouet for artichaut in courgette)
```

8.3.35 GetitemNode

A node representing a 'get item' access on a python object, in other words the '[stuff]' in 'some_object[stuff]'.

```
In [266]: RedBaron("a[b]")[0].value[1].help(deep=True)
GetitemNode()
  # identifiers: getitem, getitem_, getitemnode
  value ->
    NameNode()
      # identifiers: name, name_, namenode
      value='b'
```

SetAttr

```
In [267]: red = RedBaron("a[b]")

In [268]: red
Out[268]: 0   a[b]

In [269]: red[0].value[1].value = "1 + 1"

In [270]: red
Out[270]: 0   a[1 + 1]
```

8.3.36 GlobalNode

A node representing a global statement.

```
In [271]: RedBaron("global a")[0].help(deep=True)
GlobalNode()
  # identifiers: global, global_, globalnode
  value ->
    * NameNode()
        # identifiers: name, name_, namenode
        value='a'
```

SetAttr

```
In [272]: red = RedBaron("global a")

In [273]: red
Out[273]: 0   global a

In [274]: red[0].value = "stuff, plop"

In [275]: red
Out[275]: 0   global stuff, plop
```

8.3.37 IfNode

A node representing an if statement.

The IfNode, like the *ElifNode* or the *ElseNode*, is stored in an *IfelseblockNode*.

```
In [276]: RedBaron("if a: pass")[0].value[0].help(deep=True)
IfNode()
  # identifiers: if, if_, ifnode
  test ->
    NameNode()
      # identifiers: name, name_, namenode
      value='a'
  value ->
    * PassNode()
        # identifiers: pass, pass_, passnode
```

SetAttr

IfNode is a CodeBlockNode which means its value attribute accepts a wide range of values, see *CodeBlockNode* for more information. Other attributes work as expected:

```
In [277]: red = RedBaron("if a: pass")

In [278]: red
Out[278]:
0   if a: pass


In [279]: red[0].value[0].test = "1 + 1 == 11"

In [280]: red
```

(continues on next page)

(continued from previous page)

```
Out[280]:
0   if 1 + 1 == 11: pass
```

8.3.38 IfelseblockNode

A node representing the conditional block composed of at least one if statement, zero or more elif statements and, at the end, an optional else statement. All those statements are stored in a list.

```
In [281]: RedBaron("if a: pass\nelif b: pass\nelse: pass\n")[0].help(deep=True)
IfelseblockNode()
  # identifiers: ifelseblock, ifelseblock_, ifelseblocknode
  value ->
    * IfNode()
        # identifiers: if, if_, ifnode
        test ->
          NameNode()
            # identifiers: name, name_, namenode
            value='a'
        value ->
          * PassNode()
              # identifiers: pass, pass_, passnode
    * ElifNode()
        # identifiers: elif, elif_, elifnode
        test ->
          NameNode()
            # identifiers: name, name_, namenode
            value='b'
        value ->
          * PassNode()
              # identifiers: pass, pass_, passnode
    * ElseNode()
        # identifiers: else, else_, elsenode
        value ->
          * PassNode()
              # identifiers: pass, pass_, passnode
```

SetAttr

Works as expected and is very flexible on its input:

- the input is automatically put at the correct indentation
- the input is automatically right strip
- if the statement is followed, the correct number of blanks lines are added: 2 when at the root of the file, 1 when indented

```
In [282]: red = RedBaron("if a: pass\n")

In [283]: red
Out[283]:
0   if a: pass
```

(continues on next page)

(continued from previous page)

```
In [284]: red[0].value = "if a:\n    pass\nelif b:\n    pass\n\n\n"

In [285]: red
Out[285]:
0   if a:
        pass
    elif b:
        pass


In [286]: red[0].value = "    if a:\n        pass"

In [287]: red
Out[287]:
0   if a:
        pass
```

```
In [288]: red = RedBaron("if a:\n    pass\n\n\nplop")

In [289]: red
Out[289]:
0   if a:
        pass



1   plop

In [290]: red[0].value = "    if a:\n        pass"

In [291]: red
Out[291]:
0   if a:
        pass



1   plop
```

```
In [292]: red = RedBaron("while True:\n    if plop:\n        break\n\n    stuff")

In [293]: red
Out[293]:
0   while True:
        if plop:
            break

        stuff


In [294]: red[0].value[1].value = "if a:\n    pass\nelif b:\n    pass\n\n\n"

In [295]: red
Out[295]:
0   while True:
        if plop:
```

(continues on next page)

(continued from previous page)

```
            break

        if a:
        pass
    elif b:
        pass
```

8.3.39 ImportNode

A node representing the import statement of the python language.

Be careful, this node and its subnodes are way more complex than what you can expect.

```
In [296]: RedBaron("import foo")[0].help(deep=True)
ImportNode()
  # identifiers: import, import_, importnode
  # helpers: modules, names
  value ->
    * DottedAsNameNode()
        # identifiers: dotted_as_name, dotted_as_name_, dottedasname, dottedasnamenode
        target=''
        value ->
          * NameNode()
              # identifiers: name, name_, namenode
              value='foo'

In [297]: RedBaron("import foo.bar.baz as stuff, another_thing.plop")[0].
↪help(deep=True)
ImportNode()
  # identifiers: import, import_, importnode
  # helpers: modules, names
  value ->
    * DottedAsNameNode()
        # identifiers: dotted_as_name, dotted_as_name_, dottedasname, dottedasnamenode
        target='stuff'
        value ->
          * NameNode()
              # identifiers: name, name_, namenode
              value='foo'
          * NameNode()
              # identifiers: name, name_, namenode
              value='bar'
          * NameNode()
              # identifiers: name, name_, namenode
              value='baz'
    * DottedAsNameNode()
        # identifiers: dotted_as_name, dotted_as_name_, dottedasname, dottedasnamenode
        target=''
        value ->
          * NameNode()
              # identifiers: name, name_, namenode
              value='another_thing'
          * NameNode()
              # identifiers: name, name_, namenode
              value='plop'
```

SetAttr

Works as expected:

```
In [298]: red = RedBaron("import foo")

In [299]: red[0].value = "foo.bar.baz as plop, stuff, plop.dot"

In [300]: red
Out[300]: 0   import foo.bar.baz as plop, stuff, plop.dot

In [301]: red.help(deep=True)
0 -----------------------------------------------------
ImportNode()
  # identifiers: import, import_, importnode
  # helpers: modules, names
  value ->
    * DottedAsNameNode()
        # identifiers: dotted_as_name, dotted_as_name_, dottedasname, dottedasnamenode
        target='plop'
        value ->
          * NameNode()
              # identifiers: name, name_, namenode
              value='foo'
          * NameNode()
              # identifiers: name, name_, namenode
              value='bar'
          * NameNode()
              # identifiers: name, name_, namenode
              value='baz'
    * DottedAsNameNode()
        # identifiers: dotted_as_name, dotted_as_name_, dottedasname, dottedasnamenode
        target=''
        value ->
          * NameNode()
              # identifiers: name, name_, namenode
              value='stuff'
    * DottedAsNameNode()
        # identifiers: dotted_as_name, dotted_as_name_, dottedasname, dottedasnamenode
        target=''
        value ->
          * NameNode()
              # identifiers: name, name_, namenode
              value='plop'
          * NameNode()
              # identifiers: name, name_, namenode
              value='dot'
```

Helpers

To reduce the complexity, 2 helpers method are provided:

```
In [302]: red = RedBaron("import foo.bar.baz as stuff, another_thing.plop")

In [303]: red[0].modules()  # modules imported
Out[303]: ['foo.bar.baz', 'another_thing.plop']
```

(continues on next page)

(continued from previous page)

```
In [304]: red[0].names()  # names added to the context
Out[304]: ['stuff', 'another_thing.plop']
```

8.3.40 IntNode

A python integer.

```
In [305]: RedBaron("42")[0].help()
IntNode()
  # identifiers: int, int_, intnode
  value='42'
```

8.3.41 KwargsOnlyMarkerNode

New in 0.7.

A node representing the "*" in arguments declaration to force keywords only arguments after itself.

```
In [306]: RedBaron("def a(*): pass")[0].arguments[0].help(deep=True)
KwargsOnlyMarkerNode()
  # identifiers: kwargs_only_marker, kwargs_only_marker_, kwargsonlymarker,␣
↪kwargsonlymarkernode
```

8.3.42 LambdaNode

A node representing a lambda statement.

```
In [307]: RedBaron("lambda x: y")[0].help(deep=True)
LambdaNode()
  # identifiers: lambda, lambda_, lambdanode
  value ->
    NameNode()
      # identifiers: name, name_, namenode
      value='y'
  arguments ->
    * DefArgumentNode()
        # identifiers: def_argument, def_argument_, defargument, defargumentnode
        target ->
          NameNode()
            # identifiers: name, name_, namenode
            value='x'
        annotation ->
          None
        value ->
          None
```

SetAttr

Works as expected:

```
In [308]: red = RedBaron("lambda x: y")

In [309]: red
Out[309]: 0   lambda x: y

In [310]: red[0].arguments = "a, b=c, *d, **f"

In [311]: red
Out[311]: 0   lambda a, b=c, *d, **f: y

In [312]: red[0].value = "plop"

In [313]: red
Out[313]: 0   lambda a, b=c, *d, **f: plop
```

8.3.43 ListArgumentNode

A node representing a "star argument" in a function call **or** definition.

```
In [314]: RedBaron("def a(*b): pass")[0].arguments[0].help(deep=True)
ListArgumentNode()
  # identifiers: list_argument, list_argument_, listargument, listargumentnode
  value ->
    NameNode()
      # identifiers: name, name_, namenode
      value='b'
  annotation ->
    None
```

SetAttr

Works as expected:

```
In [315]: red = RedBaron("def a(*b): pass")

In [316]: red
Out[316]:
0   def a(*b): pass


In [317]: red[0].arguments[0].value = "plop"

In [318]: red
Out[318]:
0   def a(*plop): pass
```

New in 0.9

Annotations:

```
In [319]: red = RedBaron("def a(*b): pass")

In [320]: red
Out[320]:
```

(continues on next page)

(continued from previous page)

```
0   def a(*b): pass


In [321]: red[0].arguments[0].annotation = "Int"

In [322]: red
Out[322]:
0   def a(*b : Int): pass


In [323]: red[0].arguments[0].annotation
Out[323]: Int

In [324]: red
Out[324]:
0   def a(*b : Int): pass
```

8.3.44 ListComprehensionNode

A node representing a list comprehension node.

```
In [325]: RedBaron("[x for y in z]")[0].help(deep=True)
ListComprehensionNode()
  # identifiers: list_comprehension, list_comprehension_, listcomprehension,␣
↪listcomprehensionnode
  result ->
    NameNode()
      # identifiers: name, name_, namenode
      value='x'
  generators ->
    * ComprehensionLoopNode()
        # identifiers: comprehension_loop, comprehension_loop_, comprehensionloop,␣
↪comprehensionloopnode
        iterator ->
          NameNode()
            # identifiers: name, name_, namenode
            value='y'
        target ->
          NameNode()
            # identifiers: name, name_, namenode
            value='z'
        ifs ->
```

SetAttr

```
In [326]: red = RedBaron("[x for y in z]")

In [327]: red
Out[327]: 0   [x for y in z]

In [328]: red[0].result = "pouet"

In [329]: red
```

(continues on next page)

(continued from previous page)

```
Out[329]: 0   [pouet for y in z]

In [330]: red[0].generators = "for artichaut in courgette"

In [331]: red
Out[331]: 0   [pouet for artichaut in courgette]
```

8.3.45 ListNode

A node representing python sugar syntactic notation for list.

```
In [332]: RedBaron("[1, 2, 3]")[0].help(deep=True)
ListNode()
  # identifiers: list, list_, listnode
  value ->
    * IntNode()
        # identifiers: int, int_, intnode
        value='1'
    * IntNode()
        # identifiers: int, int_, intnode
        value='2'
    * IntNode()
        # identifiers: int, int_, intnode
        value='3'
```

8.3.46 NameAsNameNode

A node representing an argument to the from import statement.

```
In [333]: RedBaron("from x import a as d")[0].targets[0].help(deep=True)
NameAsNameNode()
  # identifiers: name_as_name, name_as_name_, nameasname, nameasnamenode
  value='a'
  target='d'
```

SetAttr

```
In [334]: red = RedBaron("from x import a as d")

In [335]: red
Out[335]: 0   from x import a as d

In [336]: red[0].targets[0].value = "some_random_module"

In [337]: red
Out[337]: 0   from x import some_random_module as d

In [338]: red[0].targets[0].target = "stuff"

In [339]: red
Out[339]: 0   from x import some_random_module as stuff
```

8.3.47 NonlocalNode

New in 0.7.

A node representing a nonlocal statement.

```
In [340]: RedBaron("nonlocal a")[0].help(deep=True)
NonlocalNode()
  # identifiers: nonlocal, nonlocal_, nonlocalnode
  value ->
    * NameNode()
        # identifiers: name, name_, namenode
        value='a'
```

SetAttr

```
In [341]: red = RedBaron("nonlocal a")

In [342]: red
Out[342]: 0   nonlocal a

In [343]: red[0].value = "stuff, plop"

In [344]: red
Out[344]: 0   nonlocal stuff, plop
```

8.3.48 PrintNode

A node representing a print statement.

```
In [345]: RedBaron("print(stuff)")[0].help(deep=True)
PrintNode()
  # identifiers: print, print_, printnode
  destination ->
    None
  value ->
    * AssociativeParenthesisNode()
        # identifiers: associative_parenthesis, associative_parenthesis_,␣
↪associativeparenthesis, associativeparenthesisnode
        value ->
          NameNode()
            # identifiers: name, name_, namenode
            value='stuff'
```

SetAttr

```
In [346]: red = RedBaron("print(stuff)")

In [347]: red
Out[347]: 0   print(stuff)

In [348]: red[0].destination = "some_file"
```

(continues on next page)

(continued from previous page)

```
In [349]: red
Out[349]: 0   print >>some_file, (stuff)

In [350]: red[0].value = "a, b, c"

In [351]: red
Out[351]: 0   print >>some_file, a, b, c

In [352]: red[0].destination = ""

In [353]: red
Out[353]: 0   print a, b, c

In [354]: red[0].value = ""

In [355]: red
Out[355]: 0   print
```

8.3.49 RaiseNode

A node representing a raise statement.

```
In [356]: RedBaron("raise Exception(), foo, bar")[0].help(deep=True)
RaiseNode()
  # identifiers: raise, raise_, raisenode
  comma_or_from=','
  value ->
    AtomtrailersNode()
      # identifiers: atomtrailers, atomtrailers_, atomtrailersnode
      value ->
        * NameNode()
            # identifiers: name, name_, namenode
            value='Exception'
        * CallNode()
            # identifiers: call, call_, callnode
            value ->
  instance ->
    NameNode()
      # identifiers: name, name_, namenode
      value='foo'
  traceback ->
    NameNode()
      # identifiers: name, name_, namenode
      value='bar'
```

SetAttr

```
In [357]: red = RedBaron("raise stuff")

In [358]: red
Out[358]: 0   raise stuff

In [359]: red[0].value = "foo"
```

(continues on next page)

(continued from previous page)

```
In [360]: red
Out[360]: 0   raise foo

In [361]: red[0].instance = "bar"

In [362]: red
Out[362]: 0   raise foo, bar

In [363]: red[0].traceback = "baz"

In [364]: red
Out[364]: 0   raise foo, bar, baz
```

New in 0.9

How to deal with the "raise from" notation: (by default a comma is inserted to avoid breaking backward compatibility)

```
In [365]: red = RedBaron("raise stuff")

In [366]: red
Out[366]: 0   raise stuff

In [367]: red[0].instance = "foo"

In [368]: red
Out[368]: 0   raise stuff, foo

In [369]: red[0].comma_or_from = "from"

In [370]: red
Out[370]: 0   raise stuff from foo

In [371]: red[0].comma_or_from = ","

In [372]: red
Out[372]: 0   raise stuff, foo

In [373]: red[0].instance = ""

In [374]: red
Out[374]: 0   raise stuff
```

8.3.50 ReprNode

A node representing python sugar syntactic notation for repr.

```
In [375]: RedBaron("`pouet`")[0].help(deep=True)
ReprNode()
  # identifiers: repr, repr_, reprnode
  value ->
    * NameNode()
        # identifiers: name, name_, namenode
        value='pouet'
```

8.3.51 ReturnNode

A node representing a return statement.

```
In [376]: RedBaron("return stuff")[0].help(deep=True)
ReturnNode()
  # identifiers: return, return_, returnnode
  value ->
    NameNode()
      # identifiers: name, name_, namenode
      value='stuff'
```

SetAttr

```
In [377]: red = RedBaron("return stuff")

In [378]: red
Out[378]: 0   return stuff

In [379]: red[0].value = "1 + 1"

In [380]: red
Out[380]: 0   return 1 + 1

In [381]: red[0].value = ""

In [382]: red
Out[382]: 0   return
```

8.3.52 SetNode

A node representing python sugar syntactic notation for set.

```
In [383]: RedBaron("{1, 2, 3}")[0].help(deep=True)
SetNode()
  # identifiers: set, set_, setnode
  value ->
    * IntNode()
        # identifiers: int, int_, intnode
        value='1'
    * IntNode()
        # identifiers: int, int_, intnode
        value='2'
    * IntNode()
        # identifiers: int, int_, intnode
        value='3'
```

8.3.53 SetComprehensionNode

A node representing a set comprehension node.

```
In [384]: RedBaron("{x for y in z}")[0].help(deep=True)
SetComprehensionNode()
  # identifiers: set_comprehension, set_comprehension_, setcomprehension,␣
↪setcomprehensionnode
  result ->
    NameNode()
      # identifiers: name, name_, namenode
      value='x'
  generators ->
    * ComprehensionLoopNode()
        # identifiers: comprehension_loop, comprehension_loop_, comprehensionloop,␣
↪comprehensionloopnode
        iterator ->
          NameNode()
            # identifiers: name, name_, namenode
            value='y'
        target ->
          NameNode()
            # identifiers: name, name_, namenode
            value='z'
        ifs ->
```

SetAttr

```
In [385]: red = RedBaron("{x for y in z}")

In [386]: red
Out[386]: 0   {x for y in z}

In [387]: red[0].result = "pouet"

In [388]: red
Out[388]: 0   {pouet for y in z}

In [389]: red[0].generators = "for artichaut in courgette"

In [390]: red
Out[390]: 0   {pouet for artichaut in courgette}
```

8.3.54 SliceNode

A node representing a slice, the "1:2:3" that can be found in a *GetitemNode*.

```
In [391]: RedBaron("a[1:-1:2]")[0].value[1].value.help(deep=True)
SliceNode()
  # identifiers: slice, slice_, slicenode
  has_two_colons=True
  lower ->
    IntNode()
      # identifiers: int, int_, intnode
      value='1'
  upper ->
    UnitaryOperatorNode()
      # identifiers: unitary_operator, unitary_operator_, unitaryoperator,␣
↪unitaryoperatornode
```

(continues on next page)

(continued from previous page)

```
      value='-'
      target ->
        IntNode()
          # identifiers: int, int_, intnode
          value='1'
  step ->
    IntNode()
      # identifiers: int, int_, intnode
      value='2'
```

SetAttr

```
In [392]: red = RedBaron("a[1:-1:2]")

In [393]: red
Out[393]: 0   a[1:-1:2]

In [394]: red[0].value[1].value.lower = "a"

In [395]: red
Out[395]: 0   a[a:-1:2]

In [396]: red[0].value[1].value.upper = "b"

In [397]: red
Out[397]: 0   a[a:b:2]

In [398]: red[0].value[1].value.step = "stuff"

In [399]: red
Out[399]: 0   a[a:b:stuff]

In [400]: red[0].value[1].value.step = ""

In [401]: red
Out[401]: 0   a[a:b]
```

8.3.55 SpaceNode

A formatting node representing a space. You'll probably never have to deal with it except if you play with the way the file is rendered.

Those nodes will be hidden in formatting keys 99% of the time (the only exception is if it's the last node of the file).

```
In [402]: RedBaron("1 + 1")[0].first_formatting[0].help()
SpaceNode()
  # identifiers: space, space_, spacenode
  value=' '

In [403]: RedBaron("1 + 1").help()
0 -----------------------------------------------------
BinaryOperatorNode()
```

(continues on next page)

(continued from previous page)

```
  # identifiers: binary_operator, binary_operator_, binaryoperator, binaryoperatornode
  value='+'
  first ->
    IntNode()
      # identifiers: int, int_, intnode
      value='1'
  second ->
    IntNode()
      # identifiers: int, int_, intnode
      value='1'
```

8.3.56 StarExpressionNode

New in 0.9

A node representing the result of a deconstruction in an assignment.

```
In [404]: red = RedBaron("a, *b = c")

In [405]: red
Out[405]: 0   a, *b = c

In [406]: red[0].target[1].help()
StarExpressionNode()
  # identifiers: star_expression, star_expression_, starexpression, starexpressionnode
  value ->
    NameNode()
      # identifiers: name, name_, namenode
      value='b'
```

8.3.57 StringChainNode

This is a special node that handle a particular way of writing a single string in python by putting several strings one after the other while only separated by spaces or endls.

```
In [407]: RedBaron("'a' r'b' b'c'")[0].help(deep=True)
StringChainNode()
  # identifiers: string_chain, string_chain_, stringchain, stringchainnode
  value ->
    * StringNode()
        # identifiers: string, string_, stringnode
        value="'a'"
    * RawStringNode()
        # identifiers: raw_string, raw_string_, rawstring, rawstringnode
        value="r'b'"
    * BinaryStringNode()
        # identifiers: binary_string, binary_string_, binarystring, binarystringnode
        value="b'c'"
```

SetAttr

```
In [408]: red = RedBaron("'a' r'b' b'c'")

In [409]: red
Out[409]: 0   'a' r'b' b'c'

In [410]: red[0].value = "'plip' 'plop'"

In [411]: red
Out[411]: 0   'plip' 'plop'
```

8.3.58 TernaryOperatorNode

A node representing the ternary operator expression.

```
In [412]: RedBaron("a if b else c")[0].help(deep=True)
TernaryOperatorNode()
  # identifiers: ternary_operator, ternary_operator_, ternaryoperator,␣
↪ternaryoperatornode
  first ->
    NameNode()
      # identifiers: name, name_, namenode
      value='a'
  value ->
    NameNode()
      # identifiers: name, name_, namenode
      value='b'
  second ->
    NameNode()
      # identifiers: name, name_, namenode
      value='c'
```

SetAttr

```
In [413]: red = RedBaron("a if b else c")

In [414]: red
Out[414]: 0   a if b else c

In [415]: red[0].value = "some_test"

In [416]: red
Out[416]: 0   a if some_test else c

In [417]: red[0].first = "a_value"

In [418]: red
Out[418]: 0   a_value if some_test else c

In [419]: red[0].second = "another_value"

In [420]: red
Out[420]: 0   a_value if some_test else another_value
```

8.3.59 TryNode

A node representing a try statement. This node is responsible for holding the *ExceptNode*, *FinallyNode* and *ElseNode*.

```
In [421]: RedBaron("try: pass\nexcept FooBar: pass\nexcept Exception: pass\nelse:␣
↪pass\nfinally: pass\n")[0].help(deep=True)
TryNode()
  # identifiers: try, try_, trynode
  else ->
    ElseNode()
      # identifiers: else, else_, elsenode
      value ->
        * PassNode()
            # identifiers: pass, pass_, passnode
  finally ->
    FinallyNode()
      # identifiers: finally, finally_, finallynode
      value ->
        * PassNode()
            # identifiers: pass, pass_, passnode
  value ->
    * PassNode()
        # identifiers: pass, pass_, passnode
  excepts ->
    * ExceptNode()
        # identifiers: except, except_, exceptnode
        delimiter=''
        exception ->
          NameNode()
            # identifiers: name, name_, namenode
            value='FooBar'
        target ->
          None
        value ->
          * PassNode()
              # identifiers: pass, pass_, passnode
    * ExceptNode()
        # identifiers: except, except_, exceptnode
        delimiter=''
        exception ->
          NameNode()
            # identifiers: name, name_, namenode
            value='Exception'
        target ->
          None
        value ->
          * PassNode()
              # identifiers: pass, pass_, passnode
```

SetAttr

TryNode is a CodeBlockNode which means its value attribute accepts a wide range of values, see *CodeBlockNode* for more information. For the `else` and the `finally` and the `excepts` attributes, TryNode is very flexible on the range of inputs it can get, like for a CodeBlockNode value's attribute.

Important: Since `else` and `finally` are reserved keywords in python, you need to append a _ to those attributes name to access/modify them: `node.else_` and `node.finally_`.

```
In [422]: red = RedBaron("try:\n    pass\nexcept:\n    pass\n")

In [423]: red
Out[423]:
0   try:
        pass
    except:
        pass


In [424]: red[0].else_ = "do_stuff"

In [425]: red
Out[425]:
0   try:
        pass
    except:
        pass
    else:
        do_stuff


In [426]: red[0].else_ = "else: foobar"

In [427]: red
Out[427]:
0   try:
        pass
    except:
        pass
    else:
        foobar


In [428]: red[0].else_ = "    else:\n        badly_indented_and_trailing\n\n\n\n"

In [429]: red
Out[429]:
0   try:
        pass
    except:
        pass
    else:
        badly_indented_and_trailing


# input management of finally_ works the same way than for else_
In [430]: red[0].finally_ = "close_some_stuff"

In [431]: red
Out[431]:
0   try:
        pass
    except:
        pass
    else:
        badly_indented_and_trailing
```

(continues on next page)

(continued from previous page)

```
    finally:
        close_some_stuff


In [432]: red[0].else_ = ""

In [433]: red
Out[433]:
0   try:
        pass
    except:
        pass
    finally:
        close_some_stuff


In [434]: red[0].finally_ = ""

In [435]: red
Out[435]:
0   try:
        pass
    except:
        pass


In [436]: red[0].excepts = "except A as b:\n    pass"

In [437]: red
Out[437]:
0   try:
        pass
    except A as b:
        pass


In [438]: red[0].excepts = "except X:\n    pass\nexcept Y:\n    pass"

In [439]: red
Out[439]:
0   try:
        pass
    except X:
        pass
    except Y:
        pass


# You **CAN'T** do this red[0].excepts = "foobar"
```

8.3.60 TupleNode

A node representing python sugar syntactic notation for tuple.

```
In [440]: RedBaron("(1, 2, 3)")[0].help(deep=True)
TupleNode()
  # identifiers: tuple, tuple_, tuplenode
  with_parenthesis=True
  value ->
    * IntNode()
        # identifiers: int, int_, intnode
        value='1'
    * IntNode()
        # identifiers: int, int_, intnode
        value='2'
    * IntNode()
        # identifiers: int, int_, intnode
        value='3'
```

8.3.61 UnitaryOperatorNode

A node representing a number sign modification operator like `-2` or `+42`.

```
In [441]: RedBaron("-1")[0].help(deep=True)
UnitaryOperatorNode()
  # identifiers: unitary_operator, unitary_operator_, unitaryoperator,␣
↪unitaryoperatornode
  value='-'
  target ->
    IntNode()
      # identifiers: int, int_, intnode
      value='1'
```

SetAttr

```
In [442]: red = RedBaron("-1")

In [443]: red
Out[443]: 0   -1

In [444]: red[0].target = "42"

In [445]: red
Out[445]: 0   -42

In [446]: red[0].value = "+"

In [447]: red
Out[447]: 0   +42
```

8.3.62 YieldNode

A node representing a yield statement.

```
In [448]: RedBaron("yield 42")[0].help(deep=True)
YieldNode()
```

(continues on next page)

(continued from previous page)

```
  # identifiers: yield, yield_, yieldnode
  value ->
    IntNode()
      # identifiers: int, int_, intnode
      value='42'
```

SetAttr

```
In [449]: red = RedBaron("yield 42")

In [450]: red
Out[450]: 0   yield 42

In [451]: red[0].value = "stuff"

In [452]: red
Out[452]: 0   yield stuff

In [453]: red[0].value = ""

In [454]: red
Out[454]: 0   yield
```

8.3.63 YieldAtomNode

A node representing a yield statement surrounded by parenthesis.

```
In [455]: RedBaron("(yield 42)")[0].help(deep=True)
YieldAtomNode()
  # identifiers: yield_atom, yield_atom_, yieldatom, yieldatomnode
  value ->
    IntNode()
      # identifiers: int, int_, intnode
      value='42'
```

SetAttr

```
In [456]: red = RedBaron("(yield 42)")

In [457]: red
Out[457]: 0   (yield 42)

In [458]: red[0].value = "stuff"

In [459]: red
Out[459]: 0   (yield stuff)

In [460]: red[0].value = ""

In [461]: red
Out[461]: 0   (yield)
```

8.3.64 YieldFromNode

New in 0.7.

A node representing a "yield from" statement.

```
In [462]: RedBaron("yield from 42")[0].help(deep=True)
YieldFromNode()
  # identifiers: yield_from, yield_from_, yieldfrom, yieldfromnode
  value ->
    IntNode()
      # identifiers: int, int_, intnode
      value='42'
```

SetAttr

```
In [463]: red = RedBaron("yield from 42")

In [464]: red
Out[464]: 0   yield from 42

In [465]: red[0].value = "stuff"

In [466]: red
Out[466]: 0   yield from stuff
```

8.3.65 WhileNode

A node representing a while loop.

```
In [467]: RedBaron("while condition:\n    pass")[0].help(deep=True)
WhileNode()
  # identifiers: while, while_, whilenode
  test ->
    NameNode()
      # identifiers: name, name_, namenode
      value='condition'
  else ->
    None
  value ->
    * PassNode()
        # identifiers: pass, pass_, passnode
```

SetAttr

WhileNode is a CodeBlockNode which means its value attribute accepts a wide range of values, see *CodeBlockNode* for more information. The else attributes accept a great ranges of inputs, since `else` is a reserved python keyword, you need to access it using the `else_` attribute. Other attributes work as expected:

```
In [468]: red = RedBaron("while condition: pass")

In [469]: red
Out[469]:
```

(continues on next page)

(continued from previous page)

```
0   while condition: pass


In [470]: red[0].test = "a is not None"

In [471]: red
Out[471]:
0   while a is not None: pass


In [472]: red[0].else_ = "do_stuff"

In [473]: red
Out[473]:
0   while a is not None: pass
    else:
        do_stuff


In [474]: red[0].else_ = "else: foobar"

In [475]: red
Out[475]:
0   while a is not None: pass
    else:
        foobar


In [476]: red[0].else_ = "    else:\n        badly_indented_and_trailing\n\n\n\n"

In [477]: red
Out[477]:
0   while a is not None: pass
    else:
        badly_indented_and_trailing
```

8.3.66 WithContext

A node representing a with statement.

```
In [478]: RedBaron("with a: pass")[0].help(deep=True)
WithNode()
  # identifiers: with, with_, withnode
  async=False
  contexts ->
    * WithContextItemNode()
        # identifiers: with_context_item, with_context_item_, withcontextitem,␣
↪withcontextitemnode
        value ->
          NameNode()
            # identifiers: name, name_, namenode
            value='a'
        as ->
          None
  value ->
```

(continues on next page)

(continued from previous page)

```
    * PassNode()
        # identifiers: pass, pass_, passnode
```

8.3.67 WithContextItemNode

A node representing one of the context manager items in a with statement.

```
In [479]: RedBaron("with a as b: pass")[0].contexts[0].help(deep=True)
WithContextItemNode()
  # identifiers: with_context_item, with_context_item_, withcontextitem,␣
↪withcontextitemnode
  value ->
    NameNode()
      # identifiers: name, name_, namenode
      value='a'
  as ->
    NameNode()
      # identifiers: name, name_, namenode
      value='b'
```

SetAttr

```
In [480]: red = RedBaron("with a: pass")

In [481]: red
Out[481]:
0   with a: pass


In [482]: red[0].contexts[0].value = "plop"

In [483]: red
Out[483]:
0   with plop: pass


In [484]: red[0].contexts[0].as_ = "stuff"

In [485]: red
Out[485]:
0   with plop as stuff: pass


In [486]: red[0].contexts[0].as_ = ""

In [487]: red
Out[487]:
0   with plop: pass
```

8.3.68 WithNode

A node representing a with statement.

```
In [488]: RedBaron("with a as b, c: pass")[0].help(deep=True)
WithNode()
  # identifiers: with, with_, withnode
  async=False
  contexts ->
    * WithContextItemNode()
        # identifiers: with_context_item, with_context_item_, withcontextitem,␣
↪withcontextitemnode
        value ->
          NameNode()
            # identifiers: name, name_, namenode
            value='a'
        as ->
          NameNode()
            # identifiers: name, name_, namenode
            value='b'
    * WithContextItemNode()
        # identifiers: with_context_item, with_context_item_, withcontextitem,␣
↪withcontextitemnode
        value ->
          NameNode()
            # identifiers: name, name_, namenode
            value='c'
        as ->
          None
  value ->
    * PassNode()
        # identifiers: pass, pass_, passnode
```

SetAttr

WithNode is a CodeBlockNode which means its value attribute accepts a wide range of values, see *CodeBlockNode* for more information. Other attributes work as expected:

```
In [489]: red = RedBaron("with a: pass")

In [490]: red
Out[490]:
0   with a: pass


In [491]: red[0].contexts = "b as plop, stuff()"

In [492]: red
Out[492]:
0   with b as plop, stuff(): pass
```

New in 0.8.

Async is a boolean attribute that determine if a function is async:

```
In [493]: red =  RedBaron("with a as b: pass")

In [494]: red[0].async_
Out[494]: False
```

(continues on next page)

(continued from previous page)

```
In [495]: red[0].async_ = True

In [496]: red
Out[496]:
0   async with a as b: pass


In [497]: red[0].async_ = False

In [498]: red
Out[498]:
0   with a as b: pass
```

Warning: As of python 3.7 *async* and *await* are now reserved keywords so don't uses *red.async*, it works as expected but won't make your code forward compatible.

CHAPTER 9

Indices and tables

- genindex
- modindex
- search

Index

P

Path (*class in redbaron*), 63