pycalculator Documentation

Release 0.0

Pierre Guilmin

Jan 14, 2019

Contents

CHAPTER 1

Some useful tool functions

This module implements various tool functions.

Table of contents

1.1 color()

`pycalculator.tools.`**`color`**`(`*`string`*`)`

Add colors to string output in terminal.

Parse the given string to replace every `<symbol>` block by its corresponding ANSI escape code, following this conversion table:

symbol	ANSI escape code	meaning
`</>`	`'\x1b[0m'`	reset
`<+>`	`'\x1b[1m'`	bold
`<r>`	`'\x1b[31m'`	red
`<g>`	`'\x1b[32m'`	green
`<b>`	`'\x1b[34m'`	blue

> **Warning:** This method will not work on Windows terminal.

Note:

This method automatically adds `'\x1b[0m'` (a reset escape sequence) to the end of the returned string to preserve the terminal output state.

Parameters **string** (*str*) – String to parse and print.

Returns ANSI encoded string.

Return type *str*

Examples

```
>>> color('<r>ERROR: critical error.')
'\x1b[31mERROR: critical error.\x1b[0m'

>>> print(color('<g><+>All checks passed!'))
All checks passed! # in green and bold in the terminal
```

See also:

Wikipedia 'ANSI escape code' article

CHAPTER 2

Defining and building expression trees

This module implements various classes allowing to build an expression tree from an infix expression.

Table of contents

2.1 Operator

class pycalculator.expression_tree.**Operator**(*symbol*, *function*, *category=None*, *associativity=None*, *precedence=None*)

This class defines an operator and its characteristics (category, associativity, precedence).

symbol
String representing the operator (ex: `'+'`, `'-'`, `'max'`, ...).

Type *str*

function
Defines the action of the operator on its arguments.

Type *function*

category
Defines the operator category.

Type *{'unary', 'binary', function', None}*

associativity
Defines the operator associativity.

Type *{'left', 'right', None}*

precedence
Defines the operator precedence (low value for low precedence and high value for high precedence).

Type *int* or *None*

See also:

Wikipedia 'Operator associativity' article

Wikipedia 'Order of operations' article

__init__(*symbol*, *function*, *category=None*, *associativity=None*, *precedence=None*)
Create an Operator object.

Parameters

- **symbol** (*str*) – String representing the operator (ex: `'+'`, `'-'`, `'max'`, ...).
- **function** (*function*) – Defines the action of the operator on its arguments.
- **category** (*{'unary', 'binary', function', None}*, optional) – Defines the operator category, default = *None*.
- **associativity** (*{'left', 'right', None}*, optional) – Defines the operator associativity, default = *None*.
- **precedence** (*int* or *None*, optional) – Defines the operator precedence (low value for low precedence and high value for high precedence), default = *None*.

Examples

```
>>> # define '+', '*' and 'max' operators
>>> Operator('+', operator.add, 'binary', 'right', 2)
<Operator '+': function=<built-in function add>, category='binary',␣
↪associativity='right', precedence=2>
>>> Operator('*', operator.mul, 'binary', 'right', 3)
<Operator '*': function=<built-in function mul>, category='binary',␣
↪associativity='right', precedence=3>
>>> Operator('max', max, 'function', precedence=5)
<Operator 'max': function=<built-in function max>, category='function',␣
↪associativity=None, precedence=5>

>>> # define 'my_op' operator
>>> def my_operator(a, b, c=2, string='waffle'):
...     return (a + b + c) / len(string)
...
>>> Operator('my_op', my_operator, 'function', precedence=5)
<Operator 'my_op': function=<function my_operator at 0x...>, category=
↪'function', associativity=None, precedence=5>
```

apply(**args*, ***kwargs*)
Apply the operator function to given arguments.

Parameters

- ***args** – Arguments to give to the operator function.
- ****kwargs** – Keyworded arguments to give to the operator function.

Returns Result of the application of the operator function on the given arguments.

Return type return type of `self.function()`

Examples

```
>>> Operator('-', lambda x: -x).apply(4)        # unary operator
-4
>>> Operator('+', operator.add).apply(2, 3)     # binary operator
5
>>> Operator('max', max).apply(2, 3, -8.6, 18) # function operator
18

>>> def my_operator(a, b, c=2, string='waffle'):
...     return (a + b + c) / len(string)
>>> Operator('my_op', my_operator).apply(1, 2, c=3, string='hello')
1.2
```

2.2 ExpressionTreeNode

class pycalculator.expression_tree.**ExpressionTreeNode**(*value=None*, *children=[]*)

This class defines an expression tree node.

A node is defined by a value and some children, a node value can be:

- An operator (`Operator` object), that has a various number of children (the operator function arguments).
- A value which is an `int`, a `str`, or any other type accepted by the parent node operator function. If a node holds a value then it is a leaf, it has no children.

value

Node value, if the node has children `value` has to be an `Operator` object.

Type any: `Operator` or argument value for parent `Operator` node function or *None*

children

Node children, can be empty.

Type list of ExpressionTreeNode

__init__(*value=None*, *children=[]*)

Create an ExpressionTreeNode object.

Parameters

- **value** (any (`Operator` or argument value for parent `Operator` node function)) – Node value, if the node has children `value` has to be an `Operator` object, default = *None*.
- **children** (*list of ExpressionTreeNode*) – Node children, default = *[]*.

Examples

```
>>> # define the operation '2 + 3'
>>> ExpressionTreeNode(Operator('+', operator.add),
...                    children=[ExpressionTreeNode(2),␣
↪ExpressionTreeNode(3)])
+
```

(continues on next page)

(continued from previous page)

```
└── 2
└── 3
>>> # define the operation '1 - 5 * 4'
>>> ExpressionTreeNode(Operator('-', operator.sub),
...                    children=[ExpressionTreeNode(1),
...                              ExpressionTreeNode(Operator('*', operator.
↪mul),
...
↪children=[ExpressionTreeNode(5),
...
↪ExpressionTreeNode(4)])])
-
└── 1
└── *
    └── 5
    └── 4
```

evaluate()

Evaluate the node by recursively evaluating its children.

Returns Value yielded by the evaluation of the expression tree node.

Return type any

Examples

```
>>> ExpressionTreeNode(Operator('*', operator.mul),
...                    children=[ExpressionTreeNode(4),
↪ExpressionTreeNode(7)]).evaluate()
28
>>> ExpressionTreeNode(Operator('+', operator.add),
...                    children=[ExpressionTreeNode('Hello'),
↪ExpressionTreeNode(' world !')]).evaluate()
'Hello world !'

>>> complex_tree = ExpressionTreeNode(Operator('!', math.factorial, category=
↪'unary'),
...                    [ExpressionTreeNode(Operator('len', len, category=
↪'function'),
...                                        [ExpressionTreeNode(Operator('+', operator.add,
↪category='binary'),
...                                                            [ExpressionTreeNode('abc'),
...                                                             ExpressionTreeNode('de')])])])
>>> complex_tree
!
└── len
    └── +
        └── 'abc'
        └── 'de'
>>> print(complex_tree.get_infix())
!len(('abc' + 'de'))
>>> complex_tree.evaluate()
120
```

get_infix()

Return a string representing the node and its child as a parenthesized infix expression.

Returns Parenthesized infix expression representing the tree.

Return type *str*

Examples

```
>>> ExpressionTreeNode(Operator('+', operator.add, category='binary'),
...                    children=[ExpressionTreeNode(2),␣
↪ExpressionTreeNode(3)]).get_infix()
'(2 + 3)'
>>> ExpressionTreeNode(Operator('-', operator.sub, category='unary'),
...                    children=[ExpressionTreeNode(Operator('len', len,␣
↪category='function'),
...                                                 ␣
↪children=[ExpressionTreeNode('Hello')])]).get_infix()
"-len('Hello')"
```

is_leaf()

Return a boolean indicating if the node is a leaf or not.

Returns *True* if the node has no children, *False* otherwise.

Return type *bool*

Examples

```
>>> basic_tree = ExpressionTreeNode(Operator('+', operator.add),
...                                 children=[ExpressionTreeNode(2),␣
↪ExpressionTreeNode(3)])
>>> basic_tree.is_leaf()
False
>>> basic_tree.children[0].is_leaf()
True
```

2.3 ExpressionTreeBuilder

class pycalculator.expression_tree.**ExpressionTreeBuilder**(*expr*)

This class allows to build an expression tree from an unparenthesized well-formed infix expression.

The algorithm is based on a modified version of the Shunting-yard algorithm from Wikipedia (see See Also section).

Warning: This class doesn't support functions and unary operators yet.

operator: *dict* with {key = operator symbol: value = corresponding `Operator` object} This dictionary holds the default operators known by the expression tree builder, contains by default +, −, ⋆, / and ^.

symbols: *str* String of all the symbols that can occur in the expression.

symbols_reg: *str* Regular expression used to split the given infix expression.

expr
String representing the infix expression.

Type *str*

output_queue
Output queue of the Shunting-yard algorithm.

Type queue of `ExpressionTreeNode`

operator_stack
Operator stack of the Shunting-yard algorithm.

Type stack of `ExpressionTreeNode`

chuncks
List of symbols and values in `expr` after splitting.

Type list of *str*

tree
Expression tree corresponding build from an infix expression.

Type `ExpressionTreeNode` or *None*

See also:

Wikipedia 'Shunting-yard algorithm' article

__init__(*expr*)
Create an ExpressionTreeBuilder object.

Parameters **expr** (*str*) – String representing the infix expression.

_last_operator_on_stack()
Return the last operator on stack.

Returns Last operator on the stack.

Return type `Operator`

_operator_stack_not_empty()
Check if the operator stack is empty.

Returns Return *True* if the operator stack is empty, *False* otherwise.

Return type *bool*

_parse_expression()
Parse the given expression to split it following *ExpressionTreeBuilder.symbol*.

See also:

re.split() documentation

Examples

```
>>> x = ExpressionTreeBuilder('(1-2)*4^5')
>>> x._parse_expression()
>>> x.chuncks
['(', '1', '-', '2', ')', '*', '4', '^', '5']
```

_pop_last_operator_to_queue()
Pop the last operator on the stack to the output queue.

build(*verbose=False*)
Build an expression tree from the given infix expression.

Parameters **verbose** (*bool*) – If *True* prints the detailed state of the stacks at each stage of the build.

evaluate()
Build the tree if not built and evaluate it.

Returns Returns the type returned by the last operator called.

Return type any

CHAPTER 3

Getting and checking user input infix expression

Python Module Index

p

Index

Symbols

A

B

C

E

F

G

I

O

P

S

T

V