Python JSONPath2 Documentation

David Brown

Dec 14, 2022

Contents:

This repository contains an implementation of JSONPath XPath for JSON for the Python programming language.

CHAPTER 1

Installation

The JSONPath2 library is available through PyPi so creating a virtual environment to install is what is shown below. Please keep in mind compatibility with the Pacifica Core services.

1.1 Installation in Virtual Environment

These installation instructions are intended to work on both Windows, Linux, and Mac platforms. Please keep that in mind when following the instructions.

Please install the appropriate tested version of Python for maximum chance of success.

1.1.1 Linux and Mac Installation

```
mkdir ~/.virtualenvs
python -m virtualenv ~/.virtualenvs/pacifica
. ~/.virtualenvs/pacifica/bin/activate
pip install jsonpath2
```

1.1.2 Windows Installation

This is done using PowerShell. Please do not use Batch Command.

```
mkdir "$Env:LOCALAPPDATA\virtualenvs"
python.exe -m virtualenv "$Env:LOCALAPPDATA\virtualenvs\pacifica"
& "$Env:LOCALAPPDATA\virtualenvs\pacifica\Scripts\activate.ps1"
pip install jsonpath2
```

CHAPTER 2

Example Usage

The JSONPath2 library has several APIs available to perform JSONPath matching.

2.1 Syntax

XPath	JSONPath	Description
`/`	`$`	the root JSON value
`.`	`@`	the current JSON value
`/`	`.` or `[]`	child operator
`//`	`..`	recursive descent (depth-first search)
`*`	`*`	wildcard (all elements of a JSON array; all values of a JSON object; otherwise none)
`[]`	`[]`	subscript operator
`|`	`[,]`	union operator (for two or more subscript operators)
n/a	`[start:end:step]`	slice operator (subset of elements of a JSON array)
`[]`	`?()`	filter expression (for use with subscript operator)

JSONPath Filter Expression	Description
`$` or `@`	nested JSONPath (returns `true` if any match exists; otherwise, returns `false`)
=, `!=`, >=, <= >, <	binary operator, where left- and right-hand operands are nested JSONPaths or JSON values (returns `true` if any match exists; otherwise, returns `false`)
`and`, `or`, `not`	Boolean operator, where operands are JSONPath filter expressions
`contains`	Checks if a string contains the specified substring (case-sensitive), or an array contains the specified element.
`( ... )`	parentheses

2.2 Functions

See #14 for more information.

The syntax for a function call is the name of the function followed by the arguments in parentheses, i.e., `name(arg1, arg2, ..., argN)`, where the arguments are either JSONPaths or JSON values.

```
>>> s = '{"hello":"Hello, world!"}'
'{"hello":"Hello, world!"}'
>>> import json
>>> d = json.loads(s)
{'hello':'Hello, world!'}
>>> from jsonpath2.path import Path
>>> p = Path.parse_str('$["hello"][length()]')
<jsonpath2.path.Path object>
>>> [m.current_value for m in p.match(d)]
[13]
>>> [m.node.tojsonpath() for m in p.match(d)]
['$["hello"][length()]']
```

JavaScript Function	Signature
Array.length	`length(): int`
Array.prototype.entries	`entries(): List[Tuple[int, Any]]`
Array.prototype.keys	`keys(): List[int]`
Array.prototype.values	`values(): List[Any]`
Object.entries	`entries(): List[Tuple[str, Any]]`
Object.keys	`keys(): List[str]`
Object.values	`values(): List[Any]`
String.length	`length(): int`
String.prototype.charAt	`charAt(index: int): str`
String.prototype.substring	`substring(indexStart: int, indexEnd: Optional[int]): str`

In the above table, the type aliases (`Any`, `List`, `Optional` and `Tuple`) are defined by the `typing` module from the Python Standard Library.

2.3 Examples

Some of the examples are provided by the test suite while some have been contributed via issues.

2.3.1 Test Suite Examples

Test the jsonpath module.

class bookstore_test.**TestBookStore**(*methodName='runTest'*)

Test the bookstore from original jsonpath article.

http://goessner.net/articles/JsonPath/

setUp()

Setup the class.

test_bookstore_examples_1()
Test the bookstore example 1.

```
>>> expr = Path.parse_str('$.store.book[*].author')
>>> expr.match(self.root_value)
```

test_bookstore_examples_10()
Test the bookstore example 10.

```
>>> expr = Path.parse_str('$..book[*][?(@.author = "Nigel Rees")]')
>>> expr.match(self.root_value)
```

test_bookstore_examples_11()
Test the bookstore example 11.

```
>>> expr = Path.parse_str('$..book[*][?(@.category != "fiction")]')
>>> expr.match(self.root_value)
```

test_bookstore_examples_12()
Test the bookstore example 12.

```
>>> expr = Path.parse_str('$..book[*][?(@.price>=10)]')
>>> expr.match(self.root_value)
>>> expr = Path.parse_str('$..book[*][?(@.price>10)]')
>>> expr.match(self.root_value)
```

test_bookstore_examples_13()
Test the bookstore example 13.

```
>>> expr = Path.parse_str('$..book[*][?(@.title contains "the")]')
>>> expr.match(self.root_value)
```

test_bookstore_examples_2()
Test the bookstore example 2.

```
>>> expr = Path.parse_str('$..author')
>>> expr.match(self.root_value)
```

test_bookstore_examples_3()
Test the bookstore example 3.

```
>>> expr = Path.parse_str('$.store.*')
>>> expr.match(self.root_value)
```

test_bookstore_examples_4()
Test the bookstore example 4.

```
>>> expr = Path.parse_str('$.store..price')
>>> expr.match(self.root_value)
```

test_bookstore_examples_5()
Test the bookstore example 5.

```
>>> expr = Path.parse_str('$..book[2]')
>>> expr.match(self.root_value)
```

test_bookstore_examples_6()
Test the bookstore example 6.

```
>>> expr = Path.parse_str('$..book[-1:]')
>>> expr.match(self.root_value)
>>> expr = Path.parse_str('$..book[-1]')
>>> expr.match(self.root_value)
>>> expr = Path.parse_str('$..book[3:4:1]')
>>> expr.match(self.root_value)
```

test_bookstore_examples_7()
Test the bookstore example 7.

```
>>> expr = Path.parse_str('$..book[0,1]')
>>> expr.match(self.root_value)
>>> expr = Path.parse_str('$..book[:2]')
>>> expr.match(self.root_value)
>>> expr = Path.parse_str('$..book[:2:1]')
>>> expr.match(self.root_value)
```

test_bookstore_examples_8()
Test the bookstore example 8.

```
>>> expr = Path.parse_str('$..book[*][?(@.isbn)]')
>>> expr.match(self.root_value)
```

test_bookstore_examples_9()
Test the bookstore example 9.

```
>>> expr = Path.parse_str('$..book[*][?(@.price<=10)]')
>>> expr.match(self.root_value)
>>> expr = Path.parse_str('$..book[*][?(@.price<10)]')
>>> expr.match(self.root_value)
```

class bookstore_test.**TestExtendedBookStore**(*methodName='runTest'*)
This test extends the standard bookstore test for completness.

setUp()
Copy the original bookstore document to this class.

test_bookstore_extexample_1()
Test the bookstore example with step function.

```
>>> expr = Path.parse_str('$..book[::2]')
>>> expr.match(self.root_value)
```

test_bookstore_extexamples_2()
Test the bookstore example slice with end and multiple colons.

```
>>> expr = Path.parse_str('$..book[:2:]')
>>> expr.match(self.root_value)
```

test_bookstore_extexamples_3()
Test the bookstore example slice with start and multiple colons.

```
>>> expr = Path.parse_str('$..book[2::]')
>>> expr.match(self.root_value)
```

2.3.2 Issue Examples

Issue #19

This issue involved finding the full path to the matched attribute.

The result isn't strictly supported by the library but code examples are provided.

```
import json
import typing

from jsonpath2.node import Node
from jsonpath2.nodes.root import RootNode
from jsonpath2.nodes.subscript import SubscriptNode
from jsonpath2.nodes.terminal import TerminalNode
from jsonpath2.path import Path
from jsonpath2.subscript import Subscript

data = json.loads("""
{
    "values": [
        {"type": 1, "value": 2},
        {"type": 2, "value": 3},
        {"type": 1, "value": 10}
    ]
}
""")

path = Path.parse_str("$.values.*[?(@.type = 1)].value")

def get_subscripts(node: Node) -> typing.List[typing.List[Subscript]]:
    return get_subscripts_(node, [])

def get_subscripts_(node: Node, accumulator: typing.List[typing.List[Subscript]]) ->␣
↪typing.List[typing.List[Subscript]]:
    if isinstance(node, RootNode):
        return get_subscripts_(node.next_node, accumulator)
    elif isinstance(node, SubscriptNode):
        accumulator.append(node.subscripts)
        return get_subscripts_(node.next_node, accumulator)
    elif isinstance(node, TerminalNode):
        return accumulator

for match_data in path.match(data):
    print(f"Value: {match_data.current_value}")
    print(f"JSONPath: {match_data.node.tojsonpath()}")
    print(f"Subscripts: {get_subscripts(match_data.node)}")
    print("")
```

The snippet above iterates over the match results, prints the value and JSONPath and then prints the list of subscripts. The list of subscripts is constructed by traversing the structure of the abstract syntax tree for the JSONPath.

The results [modulo the memory addresses] are:

```
Value: 2
JSONPath: $["values"][0]["value"]
Subscripts: [[<jsonpath2.subscripts.objectindex.ObjectIndexSubscript object at␣
↪0x10f6a3278>], [<jsonpath2.subscripts.arrayindex.ArrayIndexSubscript object at␣
↪0x10f6a37b8>], [<jsonpath2.subscripts.objectindex.ObjectIndexSubscript object at␣
↪0x10f6a3390>]]
```

(continues on next page)

(continued from previous page)

```
Value: 10
JSONPath: $["values"][2]["value"]
Subscripts: [[<jsonpath2.subscripts.objectindex.ObjectIndexSubscript object at␣
↪0x10f6a3278>], [<jsonpath2.subscripts.arrayindex.ArrayIndexSubscript object at␣
↪0x10f6a3978>], [<jsonpath2.subscripts.objectindex.ObjectIndexSubscript object at␣
↪0x10f6a3390>]]
```

The first subscript is the `"values"` key. The second subscript is the index of the `{"type":"value"}` object. The third subscript is the `"value"` key.

Note that the result (the list of subscripts) is a list of lists. This is because instances of the `SubscriptNode` class are constructed using zero or more instances of the `Subscript` class.

2.4 Grammar and parser

The ANTLR v4 grammar for JSONPath is available at `jsonpath2/parser/JSONPath.g4`.

2.4.1 Installing ANTLR v4

Adapted from antlr docs.

```
cd /usr/local/lib
curl -O https://www.antlr.org/download/antlr-4.10.1-complete.jar

export CLASSPATH=".:/usr/local/lib/antlr-4.10.1-complete.jar:$CLASSPATH"

alias antlr4='java -Xmx500M -cp "/usr/local/lib/antlr-4.10.1-complete.jar:$CLASSPATH"␣
↪org.antlr.v4.Tool'
alias grun='java org.antlr.v4.gui.TestRig'
```

2.4.2 Building the parser for the grammar

Adapted from antlr docs.

```
antlr4 -Dlanguage=Python3 -o . -lib . jsonpath2/parser/JSONPath.g4
```

CHAPTER 3

Python JSONPath2 Module

3.1 Python JSONPath2 Expressions Module

Expressions used in jsonpath module.

The operator expression module.

class jsonpath2.expressions.operator.**AndVariadicOperatorExpression**(**args*, ***kwargs*)

The boolean 'and' operator expression.

__init__(**args*, ***kwargs*)
Call the super with the 'and' boolean method.

_abc_impl = <_abc_data object>

class jsonpath2.expressions.operator.**BinaryOperatorExpression**(*token: str, callback: Callable[[object, object], bool], left_node_or_value: Union[jsonpath2.node.Node, object], right_node_or_value: Union[jsonpath2.node.Node, object]*)

Binary operator expression.

__init__(*token: str, callback: Callable[[object, object], bool], left_node_or_value: Union[jsonpath2.node.Node, object], right_node_or_value: Union[jsonpath2.node.Node, object]*)
Constructor save the left right and token.

_abc_impl = <_abc_data object>

evaluate(*root_value: object*, *current_value: object*) → bool
Evaluate the left and right values given the token.

class jsonpath2.expressions.operator.**ContainsBinaryOperatorExpression**(**args*, ***kwargs*)

Expression to handle in.

__init__(**args*, ***kwargs*)
Construct the binary operator with appropriate method.

_abc_impl = <_abc_data object>

class jsonpath2.expressions.operator.**EqualBinaryOperatorExpression**(**args*, ***kwargs*)

Binary Equal operator expression.

__init__(**args*, ***kwargs*)
Constructor with the right function.

_abc_impl = <_abc_data object>

class jsonpath2.expressions.operator.**GreaterThanBinaryOperatorExpression**(**args*, ***kwargs*)

Expression to handle greater than.

__init__(**args*, ***kwargs*)
Construct the binary operator with appropriate method.

_abc_impl = <_abc_data object>

class jsonpath2.expressions.operator.**GreaterThanOrEqualToBinaryOperatorExpression**(**args*, ***kwargs*)

Expression to handle greater than or equal.

__init__(**args*, ***kwargs*)
Construct the binary operator with appropriate method.

_abc_impl = <_abc_data object>

class jsonpath2.expressions.operator.**LessThanBinaryOperatorExpression**(**args*, ***kwargs*)

Expression to handle less than.

__init__(**args*, ***kwargs*)
Construct the binary operator with appropriate method.

_abc_impl = <_abc_data object>

class jsonpath2.expressions.operator.**LessThanOrEqualToBinaryOperatorExpression**(**args*, ***kwargs*)

Expression to handle less than or equal.

__init__(**args*, ***kwargs*)
Construct the binary operator with appropriate method.

_abc_impl = <_abc_data object>

class jsonpath2.expressions.operator.**NotEqualBinaryOperatorExpression**(**args*, ***kwargs*)

Binary Equal operator expression.

__init__(**args*, ***kwargs*)
Constructor with the right function.

_abc_impl = <_abc_data object>

class jsonpath2.expressions.operator.**NotUnaryOperatorExpression**(**args*, ***kwargs*)

Unary class to handle the 'not' expression.

__init__(**args*, ***kwargs*)
Call the unary operator expression with the right method.

_abc_impl = <_abc_data object>

class jsonpath2.expressions.operator.**OperatorExpression**
Basic operator expression object.

_abc_impl = <_abc_data object>

evaluate(*root_value: object*, *current_value: object*) → bool
Abstract method to evaluate the expression.

class jsonpath2.expressions.operator.**OrVariadicOperatorExpression**(**args*, ***kwargs*)
The boolean 'or' operator expression.

__init__(**args*, ***kwargs*)
Call the super with the 'or' boolean method.

_abc_impl = <_abc_data object>

class jsonpath2.expressions.operator.**UnaryOperatorExpression**(*token: str, callback: Callable[[bool], bool], expression: jsonpath2.expression.Expression*)
Unary operator expression base class.

__init__(*token: str, callback: Callable[[bool], bool], expression: jsonpath2.expression.Expression*)
Save the callback operator the token and expression.

_abc_impl = <_abc_data object>

evaluate(*root_value: object*, *current_value: object*) → bool
Evaluate the unary expression.

class jsonpath2.expressions.operator.**VariadicOperatorExpression**(*token: str, callback: Callable[[List[bool]], bool], expressions: List[jsonpath2.expression.Expression] = None*)
Base class to handle boolean expressions of variadic type.

__init__(*token: str, callback: Callable[[List[bool]], bool], expressions: List[jsonpath2.expression.Expression] = None*)
Save the operator token, callback and the list of expressions.

_abc_impl = <_abc_data object>

evaluate(*root_value: object*, *current_value: object*) → bool
Evaluate the expressions against the boolean callback.

Some expression module.

class jsonpath2.expressions.some.**SomeExpression**(*next_node_or_value: Union[jsonpath2.node.Node, object]*)
The some expression class.

__init__(*next_node_or_value: Union[jsonpath2.node.Node, object]*)
Save the next node.

_abc_impl = <_abc_data object>

evaluate(*root_value: object*, *current_value: object*) → bool
Evaluate the next node.

3.2 Python JSONPath2 Nodes Module

Nodes module contains all the node definitions.

The current node module.

class jsonpath2.nodes.current.**CurrentNode**(*next_node: jsonpath2.node.Node*)
Current node class to store current node info.

__init__(*next_node: jsonpath2.node.Node*)
Save the current node.

_abc_impl = <_abc_data object>

match(*root_value: object*, *current_value: object*) → Generator[jsonpath2.node.MatchData, None, None]
Match the current value and root value.

Recursive descent module.

class jsonpath2.nodes.recursivedescent.**RecursiveDescentNode**(*next_node: jsonpath2.node.Node*)
Recursive descent node class.

__init__(*next_node: jsonpath2.node.Node*)
Save the next node.

_abc_impl = <_abc_data object>

match(*root_value: object*, *current_value: object*) → Generator[jsonpath2.node.MatchData, None, None]
Match the root value with the current value.

Root node type.

class jsonpath2.nodes.root.**RootNode**(*next_node: jsonpath2.node.Node*)
Root node to start the process.

__init__(*next_node: jsonpath2.node.Node*)
Save the next node object.

_abc_impl = <_abc_data object>

match(*root_value: object*, *current_value: object*) → Generator[jsonpath2.node.MatchData, None, None]
Match the root value with the current value.

The subscript module.

class jsonpath2.nodes.subscript.**SubscriptNode**(*next_node: jsonpath2.node.Node*, *subscripts: List[jsonpath2.subscript.Subscript] = None*)
The subscript node class to handle '[]'.

__init__(*next_node: jsonpath2.node.Node*, *subscripts: List[jsonpath2.subscript.Subscript] = None*)
Save the next node and subscripts.

_abc_impl = <_abc_data object>

match(*root_value: object*, *current_value: object*) → Generator[jsonpath2.node.MatchData, None, None]
Match root value and current value for subscripts.

Terminal node object.

class jsonpath2.nodes.terminal.**TerminalNode**
Terminal node class.

_abc_impl = <_abc_data object>

match(*root_value: object*, *current_value: object*) → Generator[jsonpath2.node.MatchData, None, None]
Match a termainal node.

3.3 Python JSONPath2 Parser Module

The jsonpath parser module.

exception jsonpath2.parser.**CallableSubscriptNotFoundError**(*name*)
Callable subscript not found error.

__init__(*name*)
Initialize callable subscript not found error.

class jsonpath2.parser.**_ConsoleErrorListener**

syntaxError(*recognizer*, *offendingSymbol*, *line*, *column*, *msg*, *e*)

class jsonpath2.parser.**_JSONPathListener**(*_stack=None*)

__init__(*_stack=None*)
Initialize self. See help(type(self)) for accurate signature.

exitAndExpression(*ctx: jsonpath2.parser.JSONPathParser.JSONPathParser.AndExpressionContext*)

exitArray(*ctx: jsonpath2.parser.JSONPathParser.JSONPathParser.ArrayContext*)

exitJsonpath(*ctx: jsonpath2.parser.JSONPathParser.JSONPathParser.JsonpathContext*)

exitJsonpath_(*ctx: jsonpath2.parser.JSONPathParser.JSONPathParser.JsonpathContext*)

exitJsonpath__(*ctx: jsonpath2.parser.JSONPathParser.JSONPathParser.JsonpathContext*)

exitNotExpression(*ctx: jsonpath2.parser.JSONPathParser.JSONPathParser.NotExpressionContext*)

exitObj(*ctx: jsonpath2.parser.JSONPathParser.JSONPathParser.ObjContext*)

exitOrExpression(*ctx: jsonpath2.parser.JSONPathParser.JSONPathParser.OrExpressionContext*)

exitSliceable(*ctx: jsonpath2.parser.JSONPathParser.JSONPathParser.SliceableContext*)

exitSubscript(*ctx: jsonpath2.parser.JSONPathParser.JSONPathParser.SubscriptContext*)

exitSubscriptable(*ctx: jsonpath2.parser.JSONPathParser.JSONPathParser.SubscriptableContext*)

exitSubscriptableArguments(*ctx: jsonpath2.parser.JSONPathParser.JSONPathParser.SubscriptableArgumentsContext*)

exitSubscriptableBareword(*ctx: jsonpath2.parser.JSONPathParser.JSONPathParser.SubscriptableBarewordContext*)

exitSubscriptables(*ctx: jsonpath2.parser.JSONPathParser.JSONPathParser.SubscriptablesContext*)

exitValue(*ctx: jsonpath2.parser.JSONPathParser.JSONPathParser.ValueContext*)

class jsonpath2.parser.**_JSONPathParser**(*input: antlr4.BufferedTokenStream.TokenStream*, *output: TextIO = <_io.TextIOWrapper name='<stdout>' mode='w' encoding='UTF-8'>*)

tryCast(*cls*)
Override the antlr tryCast method.

jsonpath2.parser.**_createCallableSubscript**(*name*, **args*, ***kwargs*)
Create callable subscript for name, arguments and keyword arguments.

jsonpath2.parser.**_parse_input_stream**(*input_stream: antlr4.InputStream.InputStream*) → jsonpath2.nodes.root.RootNode

jsonpath2.parser.**parse_file**(**args*, ***kwargs*) → jsonpath2.nodes.root.RootNode
Parse a json path from a file.

jsonpath2.parser.**parse_str**(**args*, ***kwargs*) → jsonpath2.nodes.root.RootNode
Parse a json path from a string.

3.4 Python JSONPath2 Subscripts Module

Subscripts module contains the various subscripting classes.

Array Index subscript of the parse tree.

class jsonpath2.subscripts.arrayindex.**ArrayIndexSubscript**(*index: int*)
Array index subscript object.

__init__(*index: int*)
Save the index of the subscript.

_abc_impl = <_abc_data object>

match(*root_value: object*, *current_value: object*) → Generator[jsonpath2.node.MatchData, None, None]
Match the root value against the current value.

Array slicing module.

class jsonpath2.subscripts.arrayslice.**ArraySliceSubscript**(*start: int = None*, *end: int = None*, *step: int = None*)
Array slice class for the parse tree.

__init__(*start: int = None*, *end: int = None*, *step: int = None*)
Save the start end and step in the array slice.

_abc_impl = <_abc_data object>

match(*root_value: object*, *current_value: object*) → Generator[jsonpath2.node.MatchData, None, None]
Match an array slice between values.

Callable subscript.

class jsonpath2.subscripts.callable.**CallableSubscript**(**args*)
Callable subscript object.

__init__(**args*)
Initialize the callable subscript object.

_abc_impl = <_abc_data object>

match(*root_value: object*, *current_value: object*) → Generator[jsonpath2.node.MatchData, None, None]
Match the root value against the current value.

class jsonpath2.subscripts.callable.**CharAtCallableSubscript**(**args*)
charAt(int) callable subscript object.

_abc_impl = <_abc_data object>

class jsonpath2.subscripts.callable.**EntriesCallableSubscript**(**args*)
entries() callable subscript object.

_abc_impl = <_abc_data object>

class jsonpath2.subscripts.callable.**KeysCallableSubscript**(**args*)
keys() callable subscript object.

_abc_impl = <_abc_data object>

class jsonpath2.subscripts.callable.**LengthCallableSubscript**(**args*)
length() callable subscript object.

_abc_impl = <_abc_data object>

class jsonpath2.subscripts.callable.**SubstringCallableSubscript**(**args*)
substring(int[, int]) callable subscript object.

_abc_impl = <_abc_data object>

class jsonpath2.subscripts.callable.**ValuesCallableSubscript**(**args*)
values() callable subscript object.

_abc_impl = <_abc_data object>

Filter parse tree.

class jsonpath2.subscripts.filter.**FilterSubscript**(*expression: jsonpath2.expression.Expression*)
Filter subscript in the parse tree.

__init__(*expression: jsonpath2.expression.Expression*)
Save the filter expression.

_abc_impl = <_abc_data object>

match(*root_value: object*, *current_value: object*) → Generator[jsonpath2.node.MatchData, None, None]
Match the filter subscript against the current value.

Node.

class jsonpath2.subscripts.node.**NodeSubscript**(*next_node: jsonpath2.node.Node*)
Node subscript in the parse tree.

__init__(*next_node: jsonpath2.node.Node*)
Save the node subscript.

_abc_impl = <_abc_data object>

match(*root_value: object*, *current_value: object*) → Generator[jsonpath2.node.MatchData, None, None]
Match the node subscript against the current value.

Object index subscript module.

class jsonpath2.subscripts.objectindex.**ObjectIndexSubscript**(*index: str*)
Object index subscript part of the jsonpath parse tree.

__init__(*index: str*)
Save the string index into the json object.

_abc_impl = <_abc_data object>

match(*root_value: object*, *current_value: object*) → Generator[jsonpath2.node.MatchData, None, None]
Match the current value against the root value.

Wild cart subscript module.

class jsonpath2.subscripts.wildcard.**WildcardSubscript**
Wild card subscript part of the parse tree.

_abc_impl = <_abc_data object>

match(*root_value: object*, *current_value: object*) → Generator[jsonpath2.node.MatchData, None, None]
Match the root value against the current value.

The jsonpath2 module.

jsonpath2.**match**(*path_str: str*, *root_value: object*) → Generator[jsonpath2.node.MatchData, None, None]
Match root value of the path.

The `jsonpath2.match` function is a shortcut to match a given JSON data structure against a JSONPath string.

```
>>> import jsonpath2
>>> doc = {'hello': 'Hello, world!'}
>>> [x.current_value for x in jsonpath2.match('$.hello', doc)]
['Hello, world!']
```

Expression module.

class jsonpath2.expression.**Expression**
Add the expression methods to the jsonpath object.

_abc_impl = <_abc_data object>

evaluate(*root_value: object*, *current_value: object*) → bool
Abstract method to evaluate the expression.

The parse tree node module.

class jsonpath2.node.**MatchData**(*node*, *root_value*, *current_value*)
Match data object for storing node values.

The `jsonpath2.node.MatchData` class represents the JSON value and context for a JSONPath match.

This class is constructed with respect to a root JSON value, a current JSON value, and an abstract syntax tree node.

Attributes:

- `root_value` The root JSON value.
- `current_value` The current JSON value (i.e., the matching JSON value).
- `node` The abstract syntax tree node.

__init__(*node*, *root_value*, *current_value*)
Constructor to save root and current node values.

class jsonpath2.node.**Node**
Node object for the jsonpath parsetree.

The `jsonpath2.node.Node` class represents the abstract syntax tree for a JSONPath.

_abc_impl = <_abc_data object>

match(*root_value: object*, *current_value: object*) → Generator[jsonpath2.node.MatchData, None, None]
Abstract method to determine a node match.

Match the given root and current JSON data structures against this instance. For each match, yield an instance of the `jsonpath2.node.MatchData` class.

The path module.

class jsonpath2.path.**Path**(*root_node: jsonpath2.nodes.root.RootNode*)
Path parsetree object.

The `jsonpath2.path.Path` class represents a JSONPath.

```
>>> s = '{"hello":"Hello, world!"}'
'{"hello":"Hello, world!"}'
>>> import json
>>> d = json.loads(s)
{'hello':'Hello, world!'}
>>> from jsonpath2.path import Path
>>> p = Path.parse_str('$["hello"]')
<jsonpath2.path.Path object>
>>> [match_data.current_value for match_data in p.match(d)]
['Hello, world!']
>>> [match_data.node.tojsonpath() for match_data in p.match(d)]
['$["hello"]']
```

This class is constructed with respect to the given instance of the `jsonpath2.nodes.root.RootNode` class (viz., the `root_node` property).

Attributes:

- `root_node` The root node of the abstract syntax tree for this instance.

__init__(*root_node: jsonpath2.nodes.root.RootNode*)
Constructor saving the root node.

match(*root_value: object*) → Generator[jsonpath2.node.MatchData, None, None]
Match root value of the path.

Match the given JSON data structure against this instance. For each match, yield an instance of the `jsonpath2.node.MatchData` class.

classmethod parse_file(**args*, ***kwargs*)
A handler to parse a file.

Parse the contents of the given file and return a new instance of this class.

classmethod parse_str(**args*, ***kwargs*)
A handler to parse a string.

Parse the given string and return a new instance of this class.

The Subscript module.

class jsonpath2.subscript.**Subscript**
Subscript has no value beyond a node other than type.

_abc_impl = <_abc_data object>

match(*root_value: object*, *current_value: object*) → Generator[jsonpath2.node.MatchData, None, None]
Abstract method to determine a node match.

A JSONPath abstract class.

class jsonpath2.tojsonpath.**ToJSONPath**
Abstract class which calls internal method.

_abc_impl = <_abc_data object>

tojsonpath() → str
Get the json path from self and return it.

Returns the string representation of this instance.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

b

j

Index

Symbols

A

B

C

E

N

O

P

R

S

T

U

V

W