Booleano Documentation

Release 1.0a1

Gustavo Narea

May 10, 2017

Contents

Author Gustavo Narea.

Latest release 1.0a1

Date May 10, 2017.

Overview

Booleano is an interpreter of boolean expressions, a library to **define and run filters** available as text (e.g., in a natural language) or in Python code.

In order to handle text-based filters, Booleano ships with a fully-featured parser whose grammar is adaptive: Its properties can be overridden using simple configuration directives.

On the other hand, the library exposes a pythonic API for filters written in pure Python. These filters are particularly useful to build reusable conditions from objects provided by a third party library.

CHAPTER 1

TLDR;

a string + some variable = safe boolean evaluation

```
# is this character a minor guy with a "0" in his name and born after 1983 ?
eval_boolean(
    'age < const:majority & "o" in name & birthdate > "1983-02-02"',
    {"name": "sokka", "age": 15, "birthdate": datetime.date(1984, 1, 1)},
    {'majority': 18},
    grammar_tokens={'belongs_to': 'in'}
) => True
```

CHAPTER 2

The Fun Use Case

Booleano allow to safely evaluate an expression into something usable.

- `user:name is "john" and user:surname in {"doe", "shepard"}`

+

- `{"user": {"name": "katara", "surname"}}` => False
- `{"user": {"name": "john", "doe"}}` => True

with some code, you can provide any type you want, and the expression can still be in text:

- `user:birthdate > "03-07-1987"`
- `duration > 1m30s`

check the sample dirrectory to view more running examples !

CHAPTER 3

The Three Use Cases

Booleano has been designed to address the following use cases:

1. *Convert text-based conditions*: When you need to turn a condition available as plain text into something else (i.e., another filter).
2. *Evaluate text-based conditions*: When you have a condition available as plain text and need to iterate over items in order to filter out those for which the evaluation of the condition is not successful.
3. *Evaluate Python-based conditions*: When you have a condition represented by a Python object (nothing to be parsed) and need to iterate over items in order to filter out those for which the evaluation of the condition is not successful.

Convert text-based conditions

Say you have an online bookstore and you want your book search engine to support advanced queries that are hard to create with forms. You can have Booleano convert your users' query expressions into safe SQL *WHERE* clauses.

For example, if an user enters:

```
"python" in title and category == "computing" and (rating > 3 or publication_
↪date:year => 2007) and (not software or software:works_on_linux)
```

Booleano will parse that expression and will use a converter (defined by you) to turn the resulting parse tree into a *WHERE* clause which filters the books that meet all the requirements below:

- The book title contains the word "python".
- The book falls into the category "computing".
- The book has an average rating greater than 3 **or** it was published after 2006.
- If the books ships with software (e.g., in a CD), the software must work under Linux.

Of course, Booleano could also handle a simpler expression.

Note: The conversion result doesn't have to be text too, it can be any Python object. For example, if you use SQLAlchemy, your converter can turn the parse trees into SQLAlchemy filters.

Evaluate text-based conditions

Say you've created an alternative to the Unix utility find, but unlike **find**, users of your application don't filter results with command switches. Instead, they use boolean expressions to filter the files/directories.

For example, if an user runs the following command (where "search" is the name of your application):

```
search / 'file:extension in {"html", "htm", "xhtml"} and ("www-data" in␣
↪file:owner:groups or users:current_user == file:owner)'
```

Then, Booleano will parse the expression delimited by single quotes and **search** will iterate over all the files in the filesystem. On every iteration, **search** will use the parse tree returned by Booleano and will evaluate it against the current file; if evaluation fails, the file is excluded from the results; otherwise, it's included.

With the fictitious command above, only those HTML documents that meet at least one of the following conditions are included:

- The owner of the file is a member of the group "www-data".
- The owner of the file is the user that is running the command.

Again, Booleano could also handle a simpler expression (such as `'file:type == mime_types:html'` just to filter in all the HTML documents).

Evaluate Python-based conditions

Say you're using a third party authorization library which grants access if and only if the callable you pass to it returns `True`. On the other hand, the library provides you with one Booleano variable (which is an stateless Python object) called "current_user", that represents the name of the current user.

You could build your Python-based condition like this:

```
>>> from authorization_library import current_user, set_condition
>>> condition = current_user == "foo"
>>> condition
<Equal <Variable "current_user"> <String "foo">>
>>> set_condition(condition)
```

So `condition` represents an stateless object which the authorization library uses when it needs to find what requests should be allowed or forbidden. Internally, it executes `condition` by passing all the environment variables, so all the operations inside `condition` can find if they are met or not, like this:

```
>>> environment1 = {'user': "gustavo"}
>>> environment2 = {'user': "foo"}
>>> condition(environment1)
False
>>> condition(environment2)
True
```

CHAPTER 4

Features

The general features of Booleano include:

- Supported operands: Strings, numbers, sets, variables and functions.
- Supported operations:
  - Relationals ("equals to", "not equals", "less than", "greater than", "less than or equal to" and "greater than or equal to").
  - Membership ("belongs to" and "is subset of").
  - Logical connectives: "not", "and", "xor" and "or".
- Supports Python 2.7 and python 3.4 through 3.6.
- Comprehensive unit test suite, which covers the 100% of the package.
- Freedomware, released under the MIT/X License.

While the parser-specific features include:

- The operands can be bound to identifiers.
- The identifiers can be available under namespaces.
- Boolean expressions can contain any Unicode character, even in identifiers.
- It's easy to have one grammar per localization.
- The grammar is adaptive. You can customize it with simple parameters or via Pyparsing.
- Expressions can span multiple lines. Whitespace makes no difference, so they can contain tabs as well.
- No nesting level limit. Expressions can be a deep as you want.

CHAPTER 5

Documentation

Warning: Booleano is a library with his own story. it was first released by Gustavo Narea in 2009 in alpha release. but since 2009, this project was abandoned. in 2017, the Yupeek team has intended to use it for his project of auto scaling/monitoring micro service (Maiev) and thus has took the lead of the maintainig stuff. it's now on github, with CI and other stuff. compatible with python 3. this was not possible without the great stuff of Gustavo Narea, which wrote 750 tests with 100% of code covered...

the current version is a fork of the Alpha release, fully functional, but keep in mind that the current team is not the original author.

Quick tutorials

The following tutorials explain how to get started quickly with Booleano in an informal way. Each one addresses a use case, among the use cases for which the library was designed, so you don't have to read them all – Just read the one(s) you're interested in.

Convertible Parsing Tutorial

Todo

Rewrite this tutorial to explain step-by-step how to develop an small yet real project with Booleano.

Overview

Convertible parsing is when the boolean expressions should be converted into something else, most likely another filter.

What you need

For this you need a `grammar`, passed to a `convertible parse manager` and finally, your `converter class`.

Configuring the grammar

We're going to use the grammar with the default tokens, except for the "belongs to" and "is sub-set of" operators (the default tokens are "" and "", respectively, which is not easy to type in a keyboard):

```
from booleano.parser import Grammar

grammar = Grammar(belongs_to="in", is_subset="is subset of")
```

With this `grammar`, we can write expressions like:

- `"thursday" in {"monday", "tuesday", "wednesday", "thursday", "friday"}`
- `"thursday" in week_days`
- `{"thursday", "monday"} is subset of {"monday", "tuesday", "wednesday", "thursday", "friday"}`
- `{"thursday", "monday"} is subset of week_days`

Configuring the convertible parse manager

This is easy. You just need the grammar we created before:

```
from booleano.parser import ConvertibleParseManager

parse_manager = ConvertibleParseManager(grammar)
```

Defining a converter

You have to subclass `booleano.operations.converters.BaseConverter` and define its abstract methods (i.e., the node-specific converters) so they return the data type you want.

Parsing and converting expressions

That's all you need. Your module should look like this:

```
from booleano.parser import Grammar, ConvertibleParseManager
from your_project import YourCustomConverter

grammar = Grammar(belongs_to="in", is_subset="is subset of")
parse_manager = ConvertibleParseManager(grammar)
converter = YourCustomConverter()
```

It's now time to put out parser to the test! Let's start by checking how expressions are parsed:

```
>>> parse_manager.parse('"thursday" in {"monday", "tuesday", "wednesday", "thursday",
↪"friday"}')
<Parse tree (convertible) <BelongsTo <Set <String "monday">, <String "tuesday">,
↪<String "friday">, <String "thursday">, <String "wednesday">> <String "thursday">>>
>>> parse_manager.parse('today == "2009-07-17"')
<Parse tree (convertible) <Equal <Placeholder variable "today"> <String "2009-07-17">>
↪>
>>> parse_manager.parse('today != "2009-07-17"')
<Parse tree (convertible) <NotEqual <Placeholder variable "today"> <String "2009-07-17
↪">>>
>>> parse_manager.parse('~ today == "2009-07-17"')
<Parse tree (convertible) <Not <Equal <Placeholder variable "today"> <String "2009-07-
↪17">>>>
>>> parse_manager.parse('today > "2009-07-17"')
<Parse tree (convertible) <GreaterThan <Placeholder variable "today"> <String "2009-
↪07-17">>>
>>> parse_manager.parse('time:today == "sunday" & ~weather:will_it_rain_today("paris")
↪')
<Parse tree (convertible) <And <Equal <Placeholder variable "today" at namespace="time
↪"> <String "sunday">> <Not <Placeholder function call "will_it_rain_today"(<String
↪"paris">) at namespace="weather">>>>
```

OK, it seems like all the expressions above were parsed as expected.

In order to convert these trees with `YourCustomConverter`, you'd just need to pass its instance `converter` to the parse tree (which is a callable). For example:

```
>>> parse_tree = parse_manager.parse('today > "2009-07-17"')
>>> the_conversion_result = parse_tree(converter)
```

And `the_conversion_result` will be, well, the conversion result.

Evaluable Parsing Tutorial

Overview

this tutorial will show how to build a parser step by step, and then use it to check for a set of data, if the statment is true of false

What you need

You need an `evaluable parse manager` configured with a `symbol table` and at least one `grammar`, as well as define all the Booleano variables and functions that may be used in the expressions.

Defining your Grammar

the Grammar is the set of word used to understant your boolean expression. the default Grammar use the mathematical symbol, like the and=&, or=|, belongs_to=

to know all default Grammar tokens:

```
Grammar().get_all_tokens()
```

this retrieve all token from a default Grammar.

to customize some of the token, you just create your Grammar with the token as keyword:

```
from booleano.parser.grammar import Grammar
my_custom_grammar = Grammar(belongs_to='in')
my_other_custom_grammar = Grammar(**{'and': 'and', 'or': 'or', 'belongs_to': 'in'})
```

with this grammar, you can now write `{1} in {1, 2, 3} and name == "katara"` instead of `{1}  {1, 2, 3} & name == "katara"`

Build your symbol table

to allow the parser to validate your synthax, you must provide a set of symbol specific for your expression. think of it as a set of variables defined in advances.

your symbol tables can be some Constants, a Variable or a Function.

the creation of a SymbolTable require a name, a set of bound symbols, and optionaly some sub SymbolTables.

to bound a symbol, you must pass `Bind` instance to the SymbolTable. the first argument is the name of the object into the expression, and the 2nd argument the value of this symbol(a constant, variable or function)

```
from booleano.parser.scope import SymbolTable, Bind
from booleano.operations.operands.constants import Number, Set, String

SymbolTable('my_table', [])  # empty table
SymbolTable('my_table', [
    Bind('age', Number(14)),
    Bind('name', String("katara")),
    Bind('bending', Set({'water'})),
])  # a table with bound constants

SymbolTable(
    'my_table',
    [Bind('age', Number(14)),],
    SymbolTable('my_sub_table', [])  # this is an empty sub table
)
```

Constants

the Available Constants are Number, String and Set. all hard coded value in a expression are converted to Constants

```
from booleano.operations.operands.constants import Number, Set, String

age = Number(0)
majority_age = Number(18)

SymbolTable('my_table', [
    Bind('age', age),
    Bind('majority', majority_age),
    Bind('name', String("katara")),
    Bind('bending', Set({'water'})),
])

# this expression is valid with the folowing SymbolTable:
# age > majority & name == "katara" & {"water"}  bending
```

Variables

remember that all beelean expression will be executed with a context. you can build a parsed expression with some variables instead of constants. the Variables will use the context to resolve itself with a final value.

booleano ship with subclass of Variable that take the name of the variable into the context, and will resolve to it's value at execution.

the Available Variables are :

- `NumberVariable`: support operation valid for a number
- `BooleanVariable`: support operation for a boolean
- `StringVariable`: support pythonic operation for a String
- `SetVariable`: support operation for a set
- `DurationVariable`: support operation for a datetime.timedelat. can be compared with a string if the format is "WW days XX hours YY minutes ZZ seconds"
- `DateTimeVariable`: support operation for a datetime.datetime. can be compared with a string if the format is "%d/%m/%Y %H:%M:%S"
- `DateVariable`: support operation for a datetime.date. can be compared with a string if the format is "%d/%m/%Y"

```
from booleano.parser.scope import SymbolTable, Bind
    from booleano.operations.variables import NumberVariable, StringVariable,␣
↪SetVariable, DateVariable, DurationVariable


SymbolTable('my_table', [
    Bind('age', NumberVariable('age')),
    Bind('name', StringVariable("name")),
    Bind('bending', SetVariable("character_bendings")),
    Bind('start_training', DateVariable('training_at')),
    Bind('training_duration', DurationVariable('training_duration')),

])

# this expression is valid with the folowing SymbolTable:
# age > 14 & name == "katara" & ( { "water" }  bending |  start_training < "01-03-2017
↪" | training_duration > "3d4h")
```

Function

comming soon. documentations upgrade are welcome from cuntributor *Contributing*

Sub symbol table

SymbolTable can be nested. you can create it by adding as many sub SymbolTable at creation time (args) or via `SymbolTable.add_subtable()`. you will then access each sub tables `booleano.parsing.scope.Bind` ed value via his name folowed by the grammar token «namespace_separator» (usualy «:»)

```
from booleano.parser.scope import SymbolTable, Bind
from booleano.operations.operands.constants import Number


SymbolTable(
    'my_table',
    [
        Bind('age', Number(14)),
    ],
    SymbolTable('my_sub_table', [
        Bind('age', Number(16)),
    ])  # this is an empty sub table
)

# the folowing expression will be true:
# age == 14 & my_sub_table:age == 16
```

a subtable can have the same name as a `booleano.parsing.scope.Bind` ed variable.

```
from booleano.parser.scope import SymbolTable, Bind
from booleano.operations.operands.constants import Number

name = String("katara")

SymbolTable(
    'my_table',
    [
        Bind('character', name),
    ],
    SymbolTable('character', [
        Bind('age', Number(16)),
        Bind('name', name),
    ])
)

# the folowing expression will be true:
# character == "katara"
# character:age == 16
# character:name == "katara"
# character == "katara" & character:age == 16
```

Evaluable Parse Manager

EvaluableParseManager is the barbar name for «compiled boolean expression».

once a EvaluableParseManager is created with (Grammar + SymbolTable + expression) it can be used with different set of context, and return each time a boolean verifying the expression.

```
import datetime

from booleano.operations.operands.constants import Number
from booleano.parser.core import EvaluableParseManager
from booleano.parser.grammar import Grammar

from booleano.operations.variables import NumberVariable, StringVariable,␣
↪DurationVariable, SetVariable
```

```
from booleano.parser.scope import SymbolTable, Bind

name = StringVariable("name")

symbols = SymbolTable(
        'my_table',
        [
                Bind('character', name),
        ],
        SymbolTable('character', [
                Bind('age', NumberVariable('age')),
                Bind('name', name),
                Bind('training', DurationVariable('training')),
                Bind('bending', SetVariable('bending')),
        ]),
        SymbolTable('consts', [
                Bind('majority', Number(18)),
        ])
)

grammar = Grammar(**{'and': 'and', 'or': 'or', 'belongs_to': 'in'})


# 2: create the parser with se symbol table and the grammar (customized)
parse_manager = EvaluableParseManager(symbols, grammar)
# 3: compile a expression
compiled_expression = parse_manager.parse(
        'character:age < consts:majority & '
        '"a" in character & '
        '('
        '    {"water"} in character:bending | '
        '    character:training > "3d4h"'
        ')'
)

sample = [
        {"name": "katara", "age": 14, "training": datetime.timedelta(days=24),
↪"bending": {'water'}},
        {"name": "aang", "age": 112, "training": datetime.timedelta(days=6*365),
↪"bending": {'water', 'earth', 'fire', 'air'}},
        {"name": "zuko", "age": 16, "training": datetime.timedelta(days=29), "bending
↪": {'fire'}},
        {"name": "sokka", "age": 15, "training": datetime.timedelta(days=0), "bending
↪": set()},
]

for character in sample:
        # 4 execute the cumpiled expression with a context
        print("%s => %s" % (character, compiled_expression(character)))
```

Advanced Variable Subclasses

Once you know the Booleano functions and variables (not the same as Python functions and variables), it's time to implement them.

If, for example, you have a bookstore, you may need the variables to represent the following:

- The title of a book.
- The amount of pages in a book.
- The author(s) of a book.

You could define them as follows:

```
from booleano.operations import Variable

class BookTitle(Variable):
    """
    Booleano variable that represents the title of a book.

    """

    # We can perform equality (==, !=) and membership operations on the
    # name of a book:
    operations = set(["equality", "membership"])

    def equals(self, value, context):
        """
        Check if ``value`` is the title of the book.

        Comparison is case-insensitive.

        """
        actual_book_title = context['title'].lower()
        other_book_title = value.lower()
        return actual_book_title == other_book_title

    def belongs_to(self, value, context):
        """
        Check if word ``value`` is part of the words in the book title.

        """
        word = value.lower()
        title = context['title'].lower()
        return word in title

    def is_subset(self, value, context):
        """
        Check if the words in ``value`` are part of the words in the book
        title.

        """
        words = set(value.lower().split())
        title_words = set(context['title'].lower().split())
        return words.issubset(title_words)

    def to_python(self, value):
        return unicode(context['title'].lower())

class BookPages(Variable):
    """
    Booleano variable that represents the amount of pages in a book.

    """

    # This variable supports equality (==, !=) and inequality (<, >, <=, >=)
    # operations:
    operations = set(["equality", "inequality"])

    def equals(self, value, context):
        """
        Check that the book has the same amount of pages as ``value``.

        """
        actual_pages = context['pages']
        expected_pages = int(value)
        return actual_pages == expected_pages

    def greater_than(self, value, context):
        """
        Check that the amount of pages in the book is greater than
        ``value``.

        """
        actual_pages = context['pages']
        expected_pages = int(value)
        return actual_pages > expected_pages

    def less_than(self, value, context):
        """
        Check that the amount of pages in the book is less than ``value``.

        """
        actual_pages = context['pages']
        expected_pages = int(value)
        return actual_pages < expected_pages

    def to_python(self, value):
        return int(context['pages'])

class BookAuthor(Variable):
    """
    Booleano variable that represents the name of a book author.

    """

    # This variable only supports equality and boolean operations:
    operations = set(["equality", "boolean"])

    def __call__(self, context):
        """
        Check if the author of the book is known.

        """
        return bool(context['author'])

    def equals(self, value, context):
        """
        Check if ``value`` is the name of the book author.

        """
        expected_name = value.lower()
        actual_name = context['author'].lower()
        return expected_name == actual_name
    def to_python(self, value):
        return unicode(context['author'].lower())
```

Defining the symbol table

Once the required variables and functions have been defined, it's time give them names in the expressions:

```
from booleano.parser import SymbolTable, Bind
from booleano.operations import Number

book_title_var = BookTitle()

root_table = SymbolTable("root",
    (
    Bind("book", book_title_var),
    ),
    SymbolTable("book",
        (
        Bind("title", book_title_var),
        Bind("author", BookAuthor()),
        Bind("pages", BookPages())
        )
    ),
    SymbolTable("constants",
        (
        Bind("average_pages", Number(200)),
        )
    )
)
```

With the symbol table above, we have 5 identifiers:

- `book` and `book:title`, which are equivalent, represent the title of the book.
- `book:author` represents the name of the book author.
- `book:pages` represents the amounts of pages in the book.
- `constants:average_pages` is a named constant that represents the average amount of pages in all the books (200 in this case).

Defining the grammar

We're going to customize the tokens for the following operators:

- "~" (negation) will be "not".
- "==" ("equals") will be "is".
- "!=" ("not equals") will be "isn't".
- "" ("belongs to") will be "in".
- "" ("is sub-set of") will be "are included in".

So we instatiate it like this:

```
from booleano.parser import Grammar

new_tokens = {
    'not': "not",
    'eq': "is",
    'ne': "isn't",
    'belongs_to': "in",
    'is_subset': "are included in",
}

english_grammar = Grammar(**new_tokens)
```

Creating the parse manager

Finally, it's time to put it all together:

```
from booleano.parser import EvaluableParseManager

parse_manager = EvaluableParseManager(root_table, english_grammar)
```

Parsing and evaluating expressions

First let's check that our parser works correctly with our custom grammar:

```
>>> parse_manager.parse('book is "Programming in Ada 2005"')
<Parse tree (evaluable) <Equal <Anonymous variable [BookTitle]> <String "Programming␣
↪in Ada 2005">>>
>>> parse_manager.parse('book:title is "Programming in Ada 2005"')
<Parse tree (evaluable) <Equal <Anonymous variable [BookTitle]> <String "Programming␣
↪in Ada 2005">>>
>>> parse_manager.parse('"Programming in Ada 2005" is book')
<Parse tree (evaluable) <Equal <Anonymous variable [BookTitle]> <String "Programming␣
↪in Ada 2005">>>
>>> parse_manager.parse('book:title isn\'t "Programming in Ada 2005"')
<Parse tree (evaluable) <NotEqual <Anonymous variable [BookTitle]> <String
↪"Programming in Ada 2005">>>
>>> parse_manager.parse('{"ada", "programming"} are included in book:title')
<Parse tree (evaluable) <IsSubset <Anonymous variable [BookTitle]> <Set <String "ada">
↪, <String "programming">>>>
>>> parse_manager.parse('"software measurement" in book:title')
<Parse tree (evaluable) <BelongsTo <Anonymous variable [BookTitle]> <String "software␣
↪measurement">>>
>>> parse_manager.parse('"software engineering" in book:title & book:author is "Ian␣
↪Sommerville"')
<Parse tree (evaluable) <And <BelongsTo <Anonymous variable [BookTitle]> <String
↪"software engineering">> <Equal <Anonymous variable [BookAuthor]> <String "Ian␣
↪Sommerville">>>>
```

They all look great, so finally let's check if they are evaluated correctly too:

```
>>> books = (
... {'title': "Programming in Ada 2005", 'author': "John Barnes", 'pages': 828},
... {'title': "Software Engineering, 8th edition", 'author': "Ian Sommerville", 'pages
↪': 864},
... {'title': "Software Testing", 'author': "Ron Patton", 'pages': 408},
... )
>>> expr1 = parse_manager.parse('book is "Programming in Ada 2005"')
>>> expr1(books[0])
True
>>> expr1(books[1])
False
>>> expr1(books[2])
False
>>> expr2 = parse_manager.parse('"ron patton" is book:author')
>>> expr2(books[0])
False
>>> expr2(books[1])
False
>>> expr2(books[2])
True
>>> expr3 = parse_manager.parse('"software" in book:title')
>>> expr3(books[0])
False
>>> expr3(books[1])
True
>>> expr3(books[2])
True
>>> expr4 = parse_manager.parse('book:pages > 800')
>>> expr4(books[0])
True
>>> expr4(books[1])
True
>>> expr4(books[2])
False
```

And there you go! They were all evaluated correctly!

Supporting more than one grammar

If you have more than one language to support, that'd be a piece of cake! You can add translations in the symbol table and/or add a customized grammar.

For example, if we had to support Castilian (aka Spanish), our symbol table would've looked like this:

```
from booleano.parser import SymbolTable, Bind
from booleano.operations import Number

book_title_var = BookTitle()

root_table = SymbolTable("root",
    (
    Bind("book", book_title_var, es="libro"),
    ),
    SymbolTable("book",
        (
        Bind("title", book_title_var, es=u"título"),
        Bind("author", BookAuthor(), es="autor"),
        Bind("pages", BookPages(), es=u"páginas")
        ),
        es="libro"
    ),
    SymbolTable("constants",
        (
        Bind("average_pages", Number(200)),
        ),
        es="constantes"
    )
)
```

And we could've customized the grammar like this:

```
from booleano.parser import Grammar

new_es_tokens = {
    'not': "no",
    'eq': "es",
    'ne': "no es",
    'belongs_to': u"está en",
    'is_subset': u"están en",
}

castilian_grammar = Grammar(**new_es_tokens)
```

Finally, we'd have to add the new grammar to the parse manager:

```
from booleano.parser import EvaluableParseManager

parse_manager = EvaluableParseManager(root_table, english_grammar, es=castilian_
↪grammar)
```

Now test the expressions, but this time with our new localization:

```
>>> expr1 = parse_manager.parse('libro es "Programming in Ada 2005"', "es")
>>> expr1
<Parse tree (evaluable) <Equal <Anonymous variable [BookTitle]> <String "Programming␣
↪in Ada 2005">>>
>>> expr1(books[0])
True
>>> expr1(books[1])
False
>>> expr1(books[2])
False
>>> expr2 = parse_manager.parse(u'libro:páginas < 500', "es")
>>> expr2
<Parse tree (evaluable) <LessThan <Anonymous variable [BookPages]> <Number 500.0>>>
>>> expr2(books[0])
False
>>> expr2(books[1])
False
>>> expr2(books[2])
True
>>> expr3 = parse_manager.parse(u'"software" está en libro', "es")
>>> expr3
<Parse tree (evaluable) <BelongsTo <Anonymous variable [BookTitle]> <String "software
↪">>>
>>> expr3(books[0])
False
>>> expr3(books[1])
True
>>> expr3(books[2])
```

```
True
```

They worked just like the original, English expressions!

Pythonic Operations Tutorial

Todo

This will be implemented in Booleano v1.0a2. See blueprint pythonic-operations.

`booleano` API documentation

This section contains the API documentation for Booleano.

`booleano` is a namespace package in order to support third-party extensions under its namespace.

It is made up of the following packages:

`booleano.parser` – Boolean expressions parser

Grammar definition

Test utilities

Scope definition / name binding

Parse managers

A parse manager controls the parsers to be used in a single kind of expression, with one parser per supported grammar.

Parsers

Parse trees

`booleano.operations` – Boolean Operation Nodes

Operands

Constants

Classes

Note we're talking about **Booleano classes**, not Python classes. These are used in evaluable parsing.

Variables

Placeholder instances

Note we're talking about **placeholders for instances of Booleano classes**, not Python class instances. These are used in convertible parsing.

Operators

Logical connectives

Relational operators

Membership operators

Parse tree converters

`booleano.exc` – Exceptions raised by Booleano

Inheritance diagram

CHAPTER 6

About Booleano

If you want to learn more about Booleano, you'll find the following resources handy:

About Booleano

Booleano is a project started by Gustavo Narea in late April 2009, when he was working on some authorization stuff (PyACL and repoze.what 2) and the need to support user-friendly, plain text conditions arose.

it's now maintened by Yupeek for the scale rule system of Maiev.

Contributing

see *Contributing*

Coding conventions

The coding conventions for Booleano are not special at all – they are basically the same you will find in other Python projects. Most of the conventions below apply to Python files only, but some of them apply to any source code file:

- The character encoding should be UTF-8.
- Lines should not contain more than 119 characters.
- The new line character should be the one used in Unix systems (`\n`).
- Stick to the *widely* used Style Guide for Python Code and Docstring Conventions.
- The **unit test suite** for the package should cover 100% of the code and all its tests must pass, otherwise no release will be made. It won't make the package 100% bug-free (that's impossible), but at least we'll avoid regression bugs effectively and we'll be sure that a bug found will be just a not yet written test. It should be easy if you practice the Test-Driven Development methodology.

- All the public components of the package should be properly documented along with examples, so that people won't have to dive into our code to learn how to achieve what they want. This is optional in alpha releases only.

Acknowledgment

Big thank-yous go to:

- Paul McGuire, for making the awesome Pyparsing package (which powers the Booleano parser).
- Denis Spir, for his highly valuable recommendations since early in the development of this library and for making an alternate Booleano parser.

What's in a name?

The author of the library is a Venezuelan guy who enjoys naming projects with Castilian (aka Spanish) words. As you may have guessed, "booleano" is the Castilian translation for "boolean".

In case you wonder how would a native speaker pronounce it, it'd be something like "boo-leh-ah-noh".

Legal stuff (aka "boring stuff")

Except for the logo and this documentation, or unless explicitly told otherwise, any resource that is part of the Booleano project, including but not limited to source code in the Python Programming Language and its in-code documentation ("docstrings"), is available under the terms of the MIT/X License:

When you contribute software source code to the project, you accept to license your work under the terms of this license.

This documentation, on the other hand, copyright 2009 by Gustavo Narea, is available under the terms of the Creative Commons Attribution-Share Alike 3.0 Unported License. When you contribute documentation to the project, you accept to license your work under the terms of the same license.

Finally, the logo, also copyright 2009 by Gustavo Narea, is available under the terms of Creative Commons Attribution-No Derivative Works 3.0 Spain License.

Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

Types of Contributions

Report Bugs

Report bugs at https://github.com/Yupeek/booleano/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.
- maybe some fixture to reproduce the bug

Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with “bug” is open to whoever wants to implement it.

Implement Features

Look through the GitHub issues for features. Anything tagged with “feature” is open to whoever wants to implement it.

Write Documentation

booleano could always use more documentation, whether as part of the official booleano docs, in docstrings, or even on the web in blog posts, articles, and such.

Submit Feedback

The best way to send feedback is to file an issue at https://github.com/Yupeek/booleano/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

Get Started!

Ready to contribute? Here's how to set up *booleano* for local development.

1. Fork the *booleano* repo on GitHub.
2. Clone your fork locally:

```
$ git clone https://github.com/your_username_here/booleano.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv booleano
$ cd booleano/
$ pip install -r requirements.txt
$ python setup.py develop
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

```
$ flake8 src tests
$ isort src
$ tox
```

To get flake8, isort and tox, just pip install them into your virtualenv.

6. Build the doc:

```
$ python setup doc
```

check the resulting html in docs/build/html/

7. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

8. Submit a pull request through the GitHub website.

Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 2.7, 3.3, 3.4, 3.5. Check https://travis-ci.org/Yupeek/booleano/pull_requests and make sure that the tests pass for all supported Python versions.

Change log

Version 1.0 Alpha 1 (2009-07-17)

The first preview release of Booleano. All the essential functionality is ready: the parser and the operations are completely implemented. Basic documentation is available.

This is a feature-incomplete release which is aimed at potential users in order to get feedback about the shape they think Booleano should take. As a consequence, the API may change drastically in future releases.

Milestone at Launchpad.

Glossary

binding the name binding is the way your expression token will be converted to real value (constants or variables)
http://en.wikipedia.org/wiki/Name_binding

Python Module Index

Index