# Brabbel Documentation

***Release 0.4.3***

**Torsten Irländer**

October 02, 2015

Contents

CHAPTER 1

# Getting Started

## 1.1 About

Brabbel is a small expression language able to do some evaluations on a given set of values.

Brabbel is the German description for the first “language” of a baby and should emphasise the limited capabilities of the language.

## 1.2 Installation

Formbar is available as Pypi package. To install it use the following command:

```
<venv> pip install brabbel
```

The source is availble on Bitbucket. You can check of the source and install the library with the following command:

```
(venv)> hg clone https://bitbucket.org/ti/brabbel
(venv)> cd brabbel
```

## 1.3 Quickstart

Here is a short example on how to use brabbel:

```
from brabbel import Expression
expr = Expression("$foo < $bar")
values = {"foo": 1, "bar": 2}
expr.evaluate(values)
-> True
```

## 1.4 License

Brabbel is licensed with under the GNU General Public License version >= 2.

## 1.5 Authors

Torsten Irländer <torsten at irlaender dot de>

CHAPTER 2

# Language

## 2.1 Types

### 2.1.1 Number

Number in general means float and integers. If possible the given value will be casted into integer. If casting fails the value will be tried to converted into a float value.

```
>>> expr = "10"
>>> Expression(expr).evaluate()
10
>>> expr = "10.0"
>>> Expression(expr).evaluate()
10.0
>>> expr = "0"
>>> Expression(expr).evaluate()
0
>>> expr = "0.0"
>>> Expression(expr).evaluate()
0
>>> expr = "012"
>>> Expression(expr).evaluate()
12
```

### 2.1.2 String

All String are handled internally as a unicode string. Actually they will be encoded on parsing to ensure that they are unicode.

**Note:** Strings currently only have a limited subset of chars.

**BNF**

```
lquote ::= "'"
rquote ::= lquote
char   ::= a .. z | A .. Z | "  " | "-" | "_" | ":"
chars  ::= char | char chars
string ::= lquote + chars + rquote
```

**Examples**

```
>>> expr = "'Foo'"
>>> Expression(expr).evaluate()
u'Foo'
>>> expr = "'Foo Bar'"
>>> Expression(expr).evaluate()
u'Foo Bar'
```

### 2.1.3 Listings

**BNF**

```
lbr     ::= "["
rbr     ::= rbr
item    ::= string | number
items   ::= item | "," + item  items
listing ::= lbr + items + rbr
```

**Examples**

```
>>> expr = "[1, 2, 'foo', '42', 23]"
>>> Expression(expr).evaluate()
[1, 2, u'foo', u'42', 23]
```

### 2.1.4 Variables

Variables can be used as placeholder for dynamically injected values when evaluating the expression.

**BNF**

```
varsign  ::= "$"
char     ::= a .. z | A .. Z | "_"
chars    ::= char | char chars
variable ::= varsign + chars
```

**Examples**

```
>>> rule = "$foo < $bar"
>>> values = {'foo': 23, 'bar': 42}
>>> Expression(rule).evaluate(values)
True
```

The variables *$foo* and *$bar* will be replaced by the values in the values dictionary before the rule gets evaluated.

### 2.1.5 Constants

#### True

Will be converted into the Python "True" value.

```
>>> rule = "True == ($foo < $bar)"
>>> values = {'foo': 23, 'bar': 42}
>>> Expression(rule).evaluate(values)
True
```

#### False

Will be converted into the Python "False" value.

```
>>> rule = "False == ($foo > $bar)"
>>> values = {'foo': 23, 'bar': 42}
>>> Expression(rule).evaluate(values)
True
```

#### None

Will be converted into the Python "False" value.

## 2.2 Operators

The following operators are supported:

---

**Important:** In general **the operands of the operators must be of the same type!** Otherwise a TypeError will be raised. So Comparison of String and Integer values can not be done. This is escpesially important for None values. See *Handling None values*.

---

### 2.2.1 And

operator.**and_**()
: and_(a, b) – Same as a & b.

### 2.2.2 Or

operator.**or_**()
: or_(a, b) – Same as a | b.

### 2.2.3 Not

operator.**not_**()
: not_(a) – Same as not a.

### 2.2.4 ==

operator.**eq**()
: eq(a, b) – Same as a==b.

### 2.2.5 !=

operator.**ne**()
: ne(a, b) – Same as a!=b.

### 2.2.6 >

operator.**lt**()
lt(a, b) – Same as a<b.

### 2.2.7 >=

operator.**le**()
le(a, b) – Same as a<=b.

### 2.2.8 <

operator.**lt**()
lt(a, b) – Same as a<b.

### 2.2.9 <=

operator.**le**()
le(a, b) – Same as a<=b.

### 2.2.10 +

operator.**add**()
add(a, b) – Same as a + b.

### 2.2.11 -

operator.**sub**()
sub(a, b) – Same as a - b.

### 2.2.12 *

operator.**mul**()
mul(a, b) – Same as a * b.

#### 2.2.13 /

#### 2.2.14 In

### 2.3 Functions

#### 2.3.1 Bool

#### 2.3.2 Date

#### 2.3.3 Len

#### 2.3.4 Timedelta

### 2.4 Brabbel Pitfalls

Brabbel is not perfect. There are a number things where the Language might not behave as expected. This can become a pitfall in some cases so this section will list some of them. If you know more please write me an Email so I can add these here.

#### 2.4.1 Handling None values

Because Brabbel can only use the operators with operands of the same type you must take care to handle the case that some of the values in an Expression may be None. This will fail if '$foo' is None:

```
$foo < date('today')
```

Please handle possible None values this way:

```
not bool($foo) or $foo < date('today')
```

#### 2.4.2 None Constant

Currently the None constant will actually be converted into False.

CHAPTER 3

# Indices and tables

- genindex
- modindex
- search