Variables

ostephens edited this page on Dec 11, 2014 · 11 revisions

Variables available in expressions

Variables

variable name	meaning
`value`	the value of the cell in the base column of the current row; can be `null`
`row`	the current row; an object with more fields, with details below
`cells`	the cells of the current row, with fields that correspond to the column names; more details below
`cell`	the cell in the base column of the current row; an object with more fields, with details below
`recon`	the recon object of a cell returned from a reconciliation service or provider; an object with more fields, with details below
`record`	one or more rows grouped together to form a record; an object with more fields, with details below

Row

A `row` object has a few fields, which can be accessed with a dot operator or with square brackets: `row.index`, `row["index"]`, much like in Javascript.

field name	meaning
`row.index`	zero-based index of the current row
`row.cells`	the cells of the row, same as the "cells" variable above
`row.starred`	boolean, indicating if the row is starred
`row.flagged`	boolean, indicating if the row is flagged
`row.record`	the Record object containing the current row, same as the `record` variable above

Cells

The `cells` object, which can also be accessed as `row.cells`, has fields that correspond to the data column names. For example, `cells.Foo` returns a `cell` object representing the cell in the column named Foo of the current row. If the column name has spaces, use the square bracket method, e.g., `cells["Postal Code"]`.

Pages 86

Find a Page…

Architecture
Back Up OpenRefine Data
Broker Protocol
Cell Editing
Changes for 2.0
Changes for 2.1
Changes for 2.5
Changes for 2.6
Client Side Architecture
Clustering
Clustering In Depth
Column Editing
Data Sources
Developers Guide
Documentation For Developers

Show 71 more pages…

Clone this wiki locally

https://github.com/OpenRefine,

Clone in Desktop

When you need to get the value of the cells variable itself, you need .value at the end, e.g.,

```
cells["column name"].value
```

When you need to set or mass edit the values of the columns cells, then you can simply use a GREL expression within quotes, such as just

```
"San Francisco Bay"
```

Alternatively, you can use a Facet which has a built-in edit link next to each value in the Facet panel that allows you to replace large quantities of cell values in one shot.

Cell

A `cell` object has two fields

field name	meaning
`cell.value`	the value in the cell, which can be null, a string, a number, a boolean, or an error
`cell.recon`	an object encapsulating the reconciliation results for that cell

Recon

A `recon` object has a few fields

field name	meaning	deeper fields
`recon.judgment`	a string that is one of: "matched", "new", "none"	
`recon.matched`	a boolean, true iff judgment is "matched"	
`recon.match`	null, or the recon candidate that has been matched against this cell	`.id` `.name` `.type`
`recon.best`	null, or the best recon candidate	`.id` `.name` `.type` `.score`
`recon.features`	an object encapsulating reconciliation features	`.typeMatch` `.nameMatch` `.nameLevenshtein` `.nameWordDistance`
`recon.candidates`	an object encapsulating the default 3 candidates	`.id` `.name` `.type` `.score`

`recon.candidates` array can be accessed with something like:

```
forEach(cell.recon.candidates,v,v.id).join(",")
```

A recon candidate object has a few deeper fields: `id`, `name`, `type`, and `score`, whose meanings are obvious. `type` is an array of type IDs. So, the id of the best candidate can

be accessed as any one of

- `recon.best.id`
- `cell.recon.best.id`
- `row.cells["` (current column's name here) `"].recon.best.id`

You get the idea.

A `features` object has the following fields:

- `typeMatch`, `nameMatch`: booleans, indicating whether the best candidate matches the intended reconciliation type and whether the best candidate's name matches the cell's text exactly
- `nameLevenshtein`, `nameWordDistance`: numbers computed by comparing the best candidate's name with the cell's text; larger numbers mean bigger difference

Record

A `record` object encapsulates one or more rows that are grouped together. For example, the following data set has 2 records, the first grouping 2 rows and the second grouping 3 rows:

row	author	book	date
1.	Neal Stephenson	Anathem	2009
2.		Snow Crash	2000
3.	J.K. Rowling	Harry Potter and the Half-Blood Prince	2006
4.		Harry Potter and the Order of the Phoenix (Book 5)	2005
5.		Harry Potter and the Chamber of Secrets (Book 2)	2000

A `record` object has a few fields, which can be accessed with a dot operator or with square brackets: `record.index`, `record["index"]`, much like in Javascript.

field name	meaning	example
`record.index`	zero-based index of the current record	evaluating `row.record.index` on row 2 returns `0`
`record.cells`	the cells of the row	evaluating `row.record.cells.book.value` on row 2 returns `[ "Anathem", "Snow Crash" ]`
`row.fromRowIndex`	zero based index of the first row in the record	evaluating `row.record.fromRowIndex` on row 2 returns `0`
`row.toRowIndex`	index (not zero based) of the last row in the record	evaluating `row.record.toRowIndex` on row 2 returns `2`

GREL Controls

RodWhiteley edited this page on Apr 3, 2013 · 3 revisions

Controls supported by the OpenRefine Expression Language (GREL)

There are inline controls to support branching and essentially looping. They look like functions, but unlike functions, their arguments don't all get evaluated down to value before they get run. A control can decide which part of the code to execute and can affect the environment bindings. Functions, on the other hand, can't do either. Each control decides which of their arguments to evaluate to value, and how.

if(expression o, expression eTrue, expression eFalse)

Expression `o` is evaluated to a value. If that value is true, then expression `eTrue` is evaluated and the result is the value of the whole `if` expression.

Examples:

expression	result
`if("internationalization".length() > 10, "big string", "small string")`	`big string`
`if(mod(37, 2) == 0, "even", "odd")`	`odd`

with(expression o, variable v, expression e)

Evaluates expression `o` and binds its value to variable name `v`. Then evaluates expression `e` and returns that result.

expression	result
`with("european union".split(" "), a, a.length)`	`2`
`with("european union".split(" "), a, forEach(a, v, v.length()))`	`[ 8, 5 ]`
`with("european union".split(" "), a, forEach(a, v, v.length()).sum() / a.length())`	`6.5`

filter(expression a, variable v, expression test)

Evaluates expression `a` to an array. Then for each array element, binds its value to variable name `v`, evaluates expression `test` which should return a boolean. If the boolean is true, pushes `v` onto the result array.

expression	result
`filter([ 3, 4, 8, 7, 9 ], v, mod(v, 2) == 1)`	`[ 3, 7, 9 ]`

Pages 86

Find a Page…

Architecture
Back Up OpenRefine Data
Broker Protocol
Cell Editing
Changes for 2.0
Changes for 2.1
Changes for 2.5
Changes for 2.6
Client Side Architecture
Clustering
Clustering In Depth
Column Editing
Data Sources
Developers Guide
Documentation For Developers

Show 71 more pages…

Clone this wiki locally

https://github.com/OpenRefine,

Clone in Desktop

forEach(expression a, variable v, expression e)

Evaluates expression `a` to an array. Then for each array element, binds its value to variable name `v`, evaluates expression `e`, and pushes the result onto the result array.

expression	result
`forEach([ 3, 4, 8, 7, 9 ], n, mod(n, 2))`	`[ 1, 0, 0, 1, 1 ]`

forEachIndex(expression a, variable i, variable v, expression e)

Evaluates expression `a` to an array. Then for each array element, binds its index to variable `i` and its value to variable name `v`, evaluates expression `e`, and pushes the result onto the result array.

expression	result
`forEachIndex([ "anne", "ben", "cindy" ], j, name, (j + 1) + ". " + name).join(", ")`	`1. anne, 2. ben, 3. cindy`

forRange(number from, number to, number step, variable v, expression e)

Iterates over the variable `v` starting at `from`, incrementing by `step` each time while less than `to`. At each iteration, evaluates expression `e`, and pushes the result onto the result array.

forNonBlank(expression o, variable v, expression eNonBlank, expression eBlank)

Evaluates expression `o`. If it is non-blank, binds its value to variable name `v`, evaluates expression `eNonBlank` and returns the result. Otherwise (if `o` evaluates to blank), evaluates expression `eBlank` and returns that result instead.

isBlank, isNonBlank, isNull, isNotNull, isNumeric, isError

`isX(e)` : evaluates expression `e` and returns whether its value is `X`.

Examples:

expression	result
`isBlank("abc")`	`false`
`isNonBlank("abc")`	`true`
`isNull("abc")`	`false`
`isNumeric(2)`	`true`
`isError(1)`	`false`
`isError("abc")`	`false`
`isError(1 / 0)`	`true`

GREL Boolean Functions

RodWhiteley edited this page on Apr 3, 2013 · 3 revisions

Boolean functions supported by OpenRefine Expression Language (GREL)

See also: All GREL functions.

and(boolean b1, boolean b2, ...)

Logically AND two or more booleans to yield a boolean. For example, `and(1 < 3, 1 > 0)` returns `true` because both conditions are true.

or(boolean b1, boolean b2, ...)

Logically OR two or more booleans to yield a boolean. For example, `or(1 < 3, 1 > 7)` returns `true` because at least one of the conditions (the first one) is true.

not(boolean b)

Logically NOT a boolean to yield another boolean. For example, `not(1 > 7)` returns `true` because `1 > 7` itself is false.

GREL String Functions

whitni edited this page on Nov 10, 2014 · 16 revisions

String functions supported by OpenRefine Expression Language (GREL)

See also: All GREL Functions.

Basic

length(string s)

Returns the length of `s` as a number.

Testing String Characteristics

startsWith(string s, string sub)

Returns boolean indicating whether `s` starts with `sub`. For example, `startsWith("food", "foo")` returns `true`, whereas `startsWith("food", "bar")` returns `false`. You could also write the first case as `"food".startsWith("foo")`.

endsWith(string s, string sub)

Returns boolean indicating whether `s` ends with `sub`. For example, `endsWith("food", "ood")` returns `true`, whereas `endsWith("food", "odd")` returns `false`. You could also write the first case as `"food".endsWith("ood")`.

contains(string s, string sub)

Returns boolean indicating whether `s` contains `sub`. For example, `contains("food", "oo")` returns `true` whereas `contains("food", "ee")` returns `false`. You could also write the first case as `"food".contains("oo")`.

Basic String Modification

Case Conversion

toLowercase(string s)

Returns `s` converted to lowercase.

toUppercase(string s)

Returns `s` converted to uppercase.

toTitlecase(string s)

Returns `s` converted to titlecase. For example, `toTitlecase("Once upon a midnight dreary")` returns the string `Once Upon A Midnight Dreary`.

Trimming

trim(string s) and strip(string s)

Returns a copy of the string, with leading and trailing whitespace removed. For example, `trim(" island ")` returns the string `island`.

chomp(string s, string sep)

Returns a copy of `s` with `sep` removed from the end if `s` ends with `sep`; otherwise, just returns `s`. For example, `chomp("hardly", "ly")` and `chomp("hard", "ly")` both return the string `hard`.

Substring

substring(s, number from, optional number to)

Returns the substring of `s` starting from character index `from` and upto character index `to`. If `to` is omitted, it's understood as the end of the string `s`. For example, `substring("profound", 3)` returns the string `found`, and `substring("profound", 2, 4)` returns the string `of`.

Character indexes start from zero. Negative character indexes are understood as counting from the end of the string. For example, `substring("profound", 1, -1)` returns the string `rofoun`.

slice(s, number from, optional number to)

See `substring` function above.

get(o, number or string from, optional number to)

See `substring` function above.

Find and Replace

indexOf(string s, string sub)

Returns the index of `sub` first ocurring in `s` as a character index; or `-1` if `s` does not contain `sub`. For example, `indexOf("internationalization", "nation")` returns `5`, whereas `indexOf("internationalization", "world")` returns `-1`.

lastIndexOf(string s, string sub)

Returns the index of `sub` last ocurring in `s` as a character index; or `-1` if `s` does not contain `sub`. For example, `lastIndexOf("parallel", "a")` returns `3` (pointing at the second character "a").

replace(string s, string f, string r)

Returns the string obtained by replacing `f` with `r` in `s`. `f` can be a regular expression, in which case `r` can also contain capture groups declared in f.

For a simple example, `replace("The cow jumps over the moon and moos", "oo", "ee")` returns the string `The cow jumps over the meen and mees`.

More info on Regex

replaceChars(string s, string f, string r)

Returns the string obtained by replacing any character in `s` that is also in `f` with the character `r`. For example, `replaceChars("commas , and semicolons ; are separators", ",;", "**")` returns the string `commas ** and semicolons ** are separators`.

match(string s, regexp p)

Attempts to match the string `s` in its entirety against the regex pattern `p` and returns an array of capture groups. For example, `match("230.22398, 12.3480", /.*(\d\d\d\d)/)` returns an array of 1 string: `3480`. `match("230.22398, 12.3480", /.*\.(\d+).*\.(\d+)/)` returns an array of 2 strings: `22398` and `3480`.

Another `match()` example:

```
isNotNull(value.match(/(a.c)/))
```

returns True or False if value is like `abc` or `azc`, etc but would not match a value of `ac`. Remember to use Parentheses when using match() to denote the Capture Group or Groups inside your Regex expression so that you can get output when needed otherwise an empty array [] is shown for no match.

More info on Regex

String Parsing and Splitting

toNumber(o)

Returns `o` converted to a number.

split(s, sep)

Returns the array of strings obtained by splitting `s` at wherever `sep` is found in it. `sep` can be either a string or a regular expression. For example, `split("fire, water, earth, air", ",")` returns the array of 4 strings: "`fire`", "` water`", "` earth`", and "` air`". The double quotation marks are shown here only to highlight the fact that the spaces are retained.

More info on Regex

splitByLengths(string s, number n1, number n2, ...)

Returns the array of strings obtained by splitting s into substrings with the given lengths. For example, `splitByLengths("internationalization", 5, 6, 3)` returns an array of 3 strings: `inter` , `nation` , and `ali` .

smartSplit(string s, optional string sep)

Returns the array of strings obtained by splitting `s` by the separator `sep` . Handles quotes properly. Guesses tab or comma separator if `sep` is not given. Also, `value.escape('javascript')` is useful for previewing unprintable chars prior to using smartSplit.

```
smartSplit(value,"\n")  //split cell at Carriage Return or New Line char
```

splitByCharType(s)

Returns an array of strings obtained by splitting `s` into groups of consecutive characters where the characters within each group share the same unicode type, and consecutive groups differ in their unicode types.

for example the string `HenryCTaylor` will be split as follow

- `splitByCharType(value)[0]` will return H
- `splitByCharType(value)[1]` will return enry
- `splitByCharType(value)[2]` will return CT
- `splitByCharType(value)[3]` will return aylor

partition(string s, string or regex frag, optional boolean omitFragment)

Returns an array of strings `[ a , frag , b ]` where `a` is the substring within `s` before the first occurrence of `frag` in `s` , and `b` is the substring after `frag` in `s` . For example, `partition("internationalization", "nation")` returns 3 strings: `inter` , `nation` , and `alization` . If `s` does not contain `frag` , it returns an array of `[ s , "", "" ]` (the first string is the original `s` and the second and third strings are empty).

If `omitFragment` is `true` , `frag` is not returned. That is, the result is an array of only 2 elements.

Another Example using Regex with partition():

`value.partition(/c.e/)[1]` used on a cell with a value of "abcdefgh" will output `"cde"`

`"abcdefgh".partition(/d..g/)[0]` will output `"abc"` , since "abc" is the 1st part `[0]` , not the middle part `[1]` which is "defg", or the tail or last part `[-1]` which is "h".

More info on Regex

rpartition(string s, string or regex frag, optional boolean omitFragment)

Returns an array of strings `[ a , frag , b ]` where `a` is the substring within `s` before the **last** occurrence of `frag` in `s` , and `b` is the substring after `frag` in `s` . For example, `partition("parallel", "a")` returns 3 strings: `par` , `a` , and `llel` . If `s` does not contain

`frag`, it returns an array of `[ s, "", "" ]` (the first string is the original `s` and the second and third strings are empty).

More info on Regex

Encoding and Hashing

diff(o1, o2, optional string timeUnit)

For strings, returns the portion where they differ. For dates, it returns the difference in given time units.

escape(string s, string mode)

Escapes `s` in the given escaping mode: `html`, `xml`, `csv`, `url`, `javascript`.

unescape(string s, string mode)

Unescapes `s` in the given escaping mode: `html`, `xml`, `csv`, `url`, `javascript`.

md5(string s)

Returns the MD5 hash of `s`.

sha1(string s)

Returns the SHA-1 hash of `s`.

phonetic(string s, string encoding)

Returns the a phonetic encoding of `s`, string encoding to use). Example:

- phonetic(value,'doublemetaphone')
- phonetic(value,'metaphone')
- phonetic(value,'metaphone3')
- phonetic(value,'soundex')
- phonetic(value,'cologne')

reinterpret(string s, string encoder)

Returns `s` reinterpreted thru the given encoder. Supported encodings here.

fingerprint(string s)

Returns the fingerprint of `s`, a derived string that aims to be a more canonical form of it (this is mostly useful for finding clusters of strings related to the same information).

ngram(string s, number n)

Returns an array of the word ngrams of `s`.

ngramFingerprint(string s, number n)

Returns the n-gram fingerprint of `s`.

unicode(string s)

Returns an array of strings describing each character of `s` in their full unicode notation.

unicodeType(string s)

Returns an array of strings describing each character of `s` in their full unicode notation.

Freebase Specific

mqlKeyQuote(string s)

Quotes `s` so that it can be used as a Freebase key. More info: MQL key escaping

mqlKeyUnquote(string key)

Unquotes the MQL quoted string `key` back to its original form. More info: MQL key escaping

GREL Array Functions

RodWhiteley edited this page on Apr 3, 2013 · 3 revisions

Array functions supported by the OpenRefine Expression Language (GREL)

See also: All GREL functions.

length(array a)

Returns the length of array `a`.

slice(array a, number from, optional number to)

Returns the sub-array of `a` from its index `from` up to but not including index `to`. If `to` is omitted, it is understood to be the end of the array `a`. For example, `slice([ 0, 1, 2, 3, 4], 1, 3)` returns `[ 1, 2 ]`, and `slice([ 0, 1, 2, 3, 4], 1)` returns `[ 1, 2, 3, 4 ]`.

get(array a, number from, optional number to)

See `slice` function above.

reverse(array a)

Reverses array `a`. For example, `reverse([ 0, 1, 2, 3])` returns the array `[ 3, 2, 1, 0 ]`.

sort(array a)

Sorts array `a` in ascending order. For example, `sort([ 2, 1, 0, 3])` returns the array `[ 0, 1, 2, 3 ]`.

sum(array a)

Return the sum of the numbers in the array `a`. For example, `sum([ 2, 1, 0, 3])` returns `6`.

join(array a, string sep)

Returns the string obtained by joining the array `a` with the separator `sep`. For example, `join([ "foo", "bar", "baz" ], ";")` returns the string `foo;bar;baz`.

uniques(array a)

Returns array `a` with duplicates removed. For example, `uniques([ 2, 1, 2, 0, 1, 4, 3])` returns the array `[ 0, 1, 2, 3, 4 ]`.

GREL Math Functions

RodWhiteley edited this page on Apr 3, 2013 · 3 revisions

Math functions supported by the OpenRefine Expression Language (GREL)

See also: All GREL functions.

floor(number d)

Returns the floor of a number. For example, `floor(3.7)` returns `3` and `floor(-3.7)` returns `-4`.

ceil(number d)

Returns the ceiling of a number. For example, `ceil(3.7)` returns `4` and `ceil(-3.7)` returns `-3`.

round(number d)

Rounds a number to the nearest integer. For example, `round(3.7)` returns `4` and `round(-3.7)` returns `-4`.

min(number d1, number d2)

Returns the smaller of two numbers.

max(number d1, number d2)

Returns the larger of two numbers.

mod(number d1, number d2)

Returns `d1` modulus `d2`. For examples, `mod(74, 9)` returns `2`.

ln(number d)

Returns the natural log of `d`.

log(number d)

Returns the base 10 log of `d`.

exp(number d)

Returns e raised to the power of `d`.

pow(number d, number e)

Returns `d` raised to the power of `e`. For example, `pow(2, 3)` returns `8` (2 cubed) and `pow(3, 2)` returns `9` (3 squared).

The Square Root of any numeric value is expressed `pow(value, 0.5)`

sum(array a)

Returns the sum of numbers in `a`.

GREL Date Functions

Jared Howland edited this page on Apr 2, 2014 · 9 revisions

Date functions supported by the OpenRefine Expression Language (GREL)

See also: All GREL functions.

now()

Returns the current time.

toDate(o, boolean month_first / format1, format2, ...)

Returns `o` converted to a date object.

All other arguments are optional:

- `month_first` : set false if the date is formatted with the day before the month.
- `formatN` : attempt to parse the date using an ordered list of possible formats. See SimpleDateFormat for the syntax.

Examples: You can parse the cells "Nov-09" and "11/09" using

```
value.toDate('MM/yy','MMM-yy').toString('yyyy-MM')
```

For a date of the form: "1/4/2012 13:30:00" use GREL function:

```
toDate(value,"dd/mm/YYYY H:m:s")
```

toString(o, optional string format)

When `o` is a date, format specifies how to format the date.

diff(date d1, date d2, optional string timeUnit)

For dates, returns the difference in given time units.

inc(date d, number value, string unit)

Returns a date changed by the given amount in the given unit of time. Unit defaults to 'hour'.

datePart(date d, string unit)

Returns part of a date. Unit may be one of:

```
if ("hours".equals(part) || "hour".equals(part) || "h".equals(part)) {
if ("minutes".equals(part) || "minute".equals(part) || "min".equals(part))
```

Pages 86

Find a Page…

Architecture
Back Up OpenRefine Data
Broker Protocol
Cell Editing
Changes for 2.0
Changes for 2.1
Changes for 2.5
Changes for 2.6
Client Side Architecture
Clustering
Clustering In Depth
Column Editing
Data Sources
Developers Guide
Documentation For Developers
Show 71 more pages…

```
if ("seconds".equals(part) || "sec".equals(part) || "s".equals(part)) {
if ("milliseconds".equals(part) || "ms".equals(part) || "S".equals(part)) {
if ("years".equals(part) || "year".equals(part)) {
if ("months".equals(part) || "month".equals(part)) {
if ("weeks".equals(part) || "week".equals(part) || "w".equals(part)) {
if ("days".equals(part) || "day".equals(part) || "d".equals(part)) {
id ("weekday".equals(part)) {
if ("time".equals(part)) return c.getTimeInMillis();
```

GREL Other Functions

thadguidry edited this page on Oct 17, 2014 · 6 revisions

Other functions supported by the OpenRefine Expression Language (GREL)

See also: All GREL functions.

type(o)

Returns the type of `o`, such as `undefined`, `string`, `number`, etc.

hasField(o, string name)

Returns a boolean indicating whether `o` has a field called `name`. For example, `cell.hasField("value")` always returns `true`, as every cell has a `value` field.

jsonize(value)

Quotes a value as a JSON literal value.

parseJson(string s)

Parses `s` as JSON. `get` can then be used with `parseJson`, e.g.,

```
parseJson(" { 'a' : 1 } ").get("a")
```

returns 1.

To get all instances from a JSON array called "keywords" having the same object string name of "text", combine with the `forEach()` function to iterate over the array, e.g.,

JSON:

```
{
   "status":"OK",
   "url":"",
   "language":"english",
   "keywords":[
      {
         "text":"York en route",
         "relevance":"0.974363"
      },
      {
         "text":"Anthony Eden",
         "relevance":"0.814394"
      },
      {
         "text":"President Eisenhower",
         "relevance":"0.700189"
      }
   ]
```

Pages 86

Find a Page…

Architecture
Back Up OpenRefine Data
Broker Protocol
Cell Editing
Changes for 2.0
Changes for 2.1
Changes for 2.5
Changes for 2.6
Client Side Architecture
Clustering
Clustering In Depth
Column Editing
Data Sources
Developers Guide
Documentation For Developers
Show 71 more pages…

Clone this wiki locally
https://github.com/OpenRefine,
Clone in Desktop

```
}
```

GREL expression:

```
forEach(value.parseJson().keywords,v,v.text).join(":::")
```

will output like this:

```
York en route:::Anthony Eden:::President Eisenhower
```

cross(cell c, string projectName, string columnName)

Returns an array of zero or more rows in the project `projectName` for which the cells in their column `columnName` have the same content as cell `c` (similar to a lookup). Consider 2 projects with the following data:

My Address Book

friend	address
john	120 Main St.
mary	50 Broadway Ave.
anne	17 Morning Crescent

Christmas Gifts

gift	recipient
lamp	mary
clock	john

Now in the project "Christmas Gifts", we want to add a column containing the addresses of the recipients, which are in project "My Address Book". So we can invoke the command `Add column based on this column` on the "recipient" column in "Christmas Gifts" and enter this expression:

```
cell.cross("My Address Book", "friend")[0].cells["address"].value[0]
```

When that command is applied, the result is

Christmas Gifts

gift	recipient	address
lamp	mary	50 Broadway Ave.
clock	john	120 Main St.

facetCount(choiceValue, string facetExpression, string columnName)

Returns the facet count corresponding to the given choice value

gift	recipient	price
lamp	mary	20
clock	john	54
watch	amit	80
clock	claire	62

If we wanted to show how many of each gift we had given, we could use the following expression:

```
facetCount(value, "value", "gift")
```

This would then complete the table as so:

gift	recipient	price	count
lamp	mary	20	1
clock	john	54	2
watch	amit	80	1
clock	claire	62	2

Jsoup HTML parsing functions

Example usage of these functions is shown here.

parseHtml(string s)

returns a full HTML document and adds any missing closing tags.

You can then extract or `.select()` which portions of the HTML document you need for further splitting, partitioning, etc.

An example of extracting all table rows `<tr>` from a `<div>` using `parseHtml().select()` together is described more in Extract HTML attributes, text, links with integrated GREL commands.

select(Element e, String s)

returns an element from an HTML doc element using selector syntax.

Example:

```
value.parseHtml().select("div#content")[0].select("tr").toString()
```

This function can be used with most of the Jsoup selector syntax as documented here: http://jsoup.org/cookbook/extracting-data/selector-syntax

htmlAttr(Element e, String s)

returns a value from an attribute on an HTML Element.

htmlText(Element e)

returns the text from within an element (including all child elements).

innerHtml(Element e)

returns the innerHtml of an HTML element.

outerHtml(Element e)

returns the outerHtml of an HTML element. (Note: outerHtml Indicates the rendered text and HTML tags including the start and end tags, of the current element.)

ownText(Element e)

Gets the text owned by this HTML element only; does not get the combined text of all children.

Understanding Expressions

Yury Katkov edited this page on 30 Jun 2013 · 12 revisions

Understanding expressions.

Introduction

OpenRefine supports "expressions" mostly to transform existing data or to create new data based on existing data, much like how spreadsheet software supports "formulas". There are, however, significant differences between [DocumentationForUsers#Expressions OpenRefine's expressions] and typical spreadsheet formulas.

Variables

Consider this sample data set

	friend	age
1.	John Smith	28
2.	Jane Doe	33

When you invoke the Transform command on, say, column "friend" and enter an expression, OpenRefine will go through each row in the data (matching facets and filters, if any), and evaluate that expression for that row in order to obtain a result for that row. Whereas in a spreadsheet, you would need to store a different formula in each cell of that column, in OpenRefine, you only need one single expression. And that's made possible through the use of Variables, as explained next.

When evaluated on a row in the example above, the expression can access that row and the cell in the column "friend" of that row through Variables. Think of a variable as a placeholder for something else. For example, there is a variable named "value" that is the placeholder for the cell's content. When the expression is evaluated on row 1, the variable "value" will stand for "John Smith"; when evaluated on row 2, it will stand for "Jane Doe". So, if the expression is

```
value.split(" ")[1]
```

then for row 1, it will yield "Smith" and for row 2, "Doe". That expression splits against the space char found and takes the 2nd part.

IMPORTANT TIP: `[1]` is equivalent to saying "The 2nd part of an array or list" in GREL since indexing of arrays or lists in Refine actually begins with `[0]` or "The 1st part of an array or list".

Using Variables, a single expression yields different results for different rows.

Pages 86

Find a Page…

Architecture
Back Up OpenRefine Data
Broker Protocol
Cell Editing
Changes for 2.0
Changes for 2.1
Changes for 2.5
Changes for 2.6
Client Side Architecture
Clustering
Clustering In Depth
Column Editing
Data Sources
Developers Guide
Documentation For Developers
Show 71 more pages…

Clone this wiki locally
https://github.com/OpenRefi

Base Column

Note that an expression is typically based on one particular column in the data--the column whose drop-down menu is invoked. A lot of variables are created to stand for things about the cell in that "base column" of the current row on which the expression is evaluated. But there are still variables about the whole row, and through them, you can access cells in other columns.

Languages

Whereas each spreadsheet software has its own formula language, and only one language, OpenRefine is capable of supporting several languages for writing expressions. OpenRefine has its own native language called OpenRefine Expression Language (GREL), but you could also use Jython, or other languages if you install OpenRefine extensions that support them.

Where in GREL you write

```
value.split(" ")[1]
```

in Jython you would write

```
return value.split(" ")[1]
```

For that example the two languages are similar enough, but they don't have to be. On this documentation wiki, we will focus mostly on GREL. If you use another language, like Jython, please refer to its own documentation.

OpenRefine Expression Language (GREL)

Basics

GREL is designed to resemble Javascript. So you can expect these basic things to work, and know how they would work:

example	description
`value + " (approved)"`	concatenate two strings; whatever is in `value` gets converted to a string first
`value + 2.239`	add two numbers; if `value` actually holds something other than a number, this becomes a string concatenation
`value.trim().length()`	trimming leading and trailing whitespace of `value` and than take the length of the result
`value.substring(7, 10)`	take the substring of `value` from character index 7 up to and excluding character index 10
`value.substring(13)`	take the substring of `value` from character index 13 until the end of the string

Concatenation

If you're used to Excel, note that the operator for string concatenation is not `&` but `+`.

Function syntax

In OpenRefine expression language function can be invoked using either of these 2 forms:

- `functionName(arg0, arg1, ...)`
- `arg0.functionName(arg1, ...)`

The second form above is a shortcut to make expressions easier to read.

It's only syntactic sugar or shorthand, such as:

dot shorthand notation	full notation
`value.trim().length()`	`length(trim(value))`
`value.substring(7, 10)`	`substring(value, 7, 10)`
`value.substring(13)`	`substring(value, 13)`

The first argument to a function can be swapped out in front of the function to formulate the dot notation. That is easier to read as the functions occur from left to right in the order of calling, rather than in the reverse order.

The dot notation can also be used to access member fields:

example	description
`cell.value`	same as just `value` because `cell` stands for the current cell
`row.index`	index of the current row

For member fields whose names are not words (e.g., they contain spaces or other characters), use the bracket notation:

`cells["First Name"]`	access the cell in the column called "First Name" of the current row

Array syntax

Brackets can also be used to get substrings and sub-arrays, and their syntax resembles Python a bit more than Javascript:

example	description
`value[1,3]`	access the substring of `value` starting from character index 1 up to but excluding character index 3
`"internationalization"[1,3]`	return `nt`
`"internationalization"[1,-2]`	return `nternationalizati` (negative indexes are counted from the end)

Controls

GREL supports branching and looping (e.g., "if" and "for") slightly differently than

Javascript. To do branching and looping, you use GREL "controls", or sometimes also called "constructs", and their syntax is more like Excel's IF:

example	description
`if(value.length() > 10, "big string", "small string")`	if the length of what `value` stands for is great than 10 characters, then return "big string", otherwise, return "small string"
`if(mod(row.index, 2) == 0, "even", "odd")`	if the 2 modulus of the row index is zero, then output "even", otherwise, output "odd"

Thus, the `if` control has this syntax

`if (` test_condition `,` true_result `,` false_result `)`

The test_condition sub-expression is evaluated. If it yields true, then the true_result sub-expression is evaluated; otherwise, the false_result sub-expression is evaluated.

GREL doesn't support looping in the conventional meaning, as supported by `for` in Javascript. However, you can still process arrays of things using the various `for-` controls, e.g.,

`forEach("Once upon a time in Mexico".split(" "), v, v.length())`	return array of lengths of words, `[ 4, 4, 1, 4, 2, 6 ]`

The `forEach` control has this syntax

`for (` array_subexpr `,` element_var_name `,` element_subexpr `)`

The array_subexpr sub-expression is evaluated to an array. For each element in that array, element_subexpr sub-expression will be evaluated with the variable named by element_var_name standing for that element. In the example above, when `v.length()` is evaluated for the first element--the string "Once", the variable called `v` stands for that string "Once", and the sub-expression `v.length()` evaluates to `4`.

Another useful control is `with` that can be used to define a new variable. For instance, if we want to compute the average word length for the string "Once upon a time in Mexico", then we might start with

```
  forEach("Once upon a time in Mexico".split(" "), v, v.length()).sum() /
"Once upon a time in Mexico".split(" ").length()
```

We have `"Once upon a time in Mexico".split(" ")` repeated twice. To shorten that expression, we can define a variable to stand in that sub-expression's place:

```
  with("Once upon a time in Mexico".split(" "), a, forEach(a, v, v.length(
)).sum() / a.length())
```

Thus, the `with` control has this syntax

`with (` subexpr1 `,` var_name `,` subexpr2 `)`

First, the sub-expression subexpr1 is evaluated, and then a new variable called var_name is defined to stand in its place while the sub-expression subexpr2 is evaluated.