yaql Documentation

Release

OpenStack Foundation

Jun 27, 2017

Contents

CHAPTER 1

YAQL: Yet Another Query Language

YAQL (Yet Another Query Language) is an embeddable and extensible query language, that allows performing complex queries against arbitrary objects. It has a vast and comprehensive standard library of frequently used querying functions and can be extend even further with user-specified functions. YAQL is written in python and is distributed via PyPI.

Quickstart

Install the latest version of yaql:

```
pip install yaql>=1.0.0
```

Run yaql REPL:

```
yaql
```

Load a json file:

```
yaql> @load my_file.json
```

Check it loaded to current context, i.e. *$*:

```
yaql> $
```

Run some queries:

```
yaql> $.customers
...
yaql> $.customers.orders
...
yaql> $.customers.where($.age > 18)
...
yaql> $.customers.groupBy($.sex)
```

```
...
yaql> $.customers.where($.orders.len() >= 1 or name = "John")
```

Project Resources

- Official Documentation
- Project status, bugs, and blueprints are tracked on Launchpad

License

CHAPTER 2

What is YAQL

YAQL is a general purpose query language, that is designed to operate on objects of arbitrary complexity. YAQL has a large standard library of functions for filtering, grouping and aggregation of data. At the same time YAQL allows you to extend it by defining your own functions.

CHAPTER 3

Why YAQL?

So why bother and create another solution for a task, that has been addressed by many before us? Obviously because we were not satisfied with flexibility and/or quality of any existing solution. Most notably we needed a tool for json data, that would support some complex data transformations. YAQL is a pure-python library and therefore is easily embeddable in any python application. YAQL is designed to be human-readable and has a SQL-like feel and look. It is inspired in part by LINQ for .NET. Since YAQL is extensible and embeddable it makes a perfect choice for becoming the basis for your DSLs.

CHAPTER 4

Getting started with YAQL

Introduction to YAQL

YAQL (Yet Another Query Language) is an embeddable and extensible query language that allows performing complex queries against arbitrary data structures. *Embeddable* means that you can easily integrate a YAQL query processor in your code. Queries come from your DSLs (domain specific language), user input, JSON, and so on. YAQL has a vast and comprehensive standard library of functions that can be used to query data of any complexity. Also, YAQL can be extended even further with user-specified functions. YAQL is written in Python and is distributed through PyPI.

YAQL was inspired by Microsoft LINQ for Objects and its first aim is to execute expressions on the data in memory. A YAQL expression has the same role as an SQL query to databases: search and operate the data. In general, any SQL query can be transformed to a YAQL expression, but YAQL can also be used for computational statements. For example, *2 + 3*4* is a valid YAQL expression.

Moreover, in YAQL, the following operations are supported out of the box:

- Complex data queries
- Creation and transformation of lists, dicts, and arrays
- String operations
- Basic math operations
- Conditional expression
- Date and time operations (will be supported in yaql 1.1)

An interesting thing in YAQL is that everything is a function and any function can be customized or overridden. This is true even for built-in functions. YAQL cannot call any function that was not explicitly registered to be accessible by YAQL. The same is true for operators.

YAQL can be used in two different ways: as an independent CLI tool, and as a Python module.

Installation

You can install YAQL in two different ways:

1. Using PyPi:

```
pip install yaql
```

2. Using your system package manager (for example Ubuntu):

```
sudo apt-get install python-yaql
```

HowTo: Use YAQL in Python

You can operate with YAQL from Python in three easy steps:

- Create a YAQL engine
- Parse a YAQL expression
- Execute the parsed expression

Note: The engine should be created once for a set of operators and parser rules. It can be reused for all queries.

Here is an example how it can be done with the YAML file which looks like:

```
customers_city:
  - city: New York
    customer_id: 1
  - city: Saint Louis
    customer_id: 2
  - city: Mountain View
    customer_id: 3
customers:
  - customer_id: 1
    name: John
    orders:
      - order_id: 1
        item: Guitar
        quantity: 1
  - customer_id: 2
    name: Paul
    orders:
      - order_id: 2
        item: Banjo
        quantity: 2
      - order_id: 3
        item: Piano
        quantity: 1
  - customer_id: 3
    name: Diana
    orders:
      - order_id: 4
        item: Drums
        quantity: 1
```

```
import yaql
import yaml

data_source = yaml.load(open('shop.yaml', 'r'))

engine = yaql.factory.YaqlFactory().create()

expression = engine(
    '$.customers.orders.selectMany($.where($.order_id = 4))')

order = expression.evaluate(data=data_source)
```

Content of the `order` will be the following:

```
[{u'item': u'Drums', u'order_id': 4, u'quantity': 1}]
```

YAQL grammar

YAQL has a very simple grammar:

- Three keywords as in JSON: true, false, null
- Numbers, such as 12 and 34.5
- Strings: *'foo'* and *"bar"*
- Access to the data: $variable, $
- Binary and unary operators: 2 + 2, -1, 1 != 2, $list[1]

Data access

Although YAQL expressions may be self-sufficient, the most important value of YAQL is its ability to operate on user-passed data. Such data is placed into variables which are accessible in a YAQL expression as *$<variable_name>*. The *variable_name* can contain numbers, English alphabetic characters, and underscore symbols. The *variable_name* can be empty, in this case you will use *$*. Variables can be set prior to executing a YAQL expression or can be changed during the execution of some functions.

According to the convention in YAQL, function parameters, including input data, are stored in variables like *$1*, *$2*, and so on. The *$* stands for *$1*. For most cases, all function parameters are passed in one piece and can be accessed using *$*, that is why this variable is the most used one in YAQL expressions. Besides, some functions are expected to get a YAQL expression as one of the parameters (for example, a predicate for collection sorting). In this case, passed expression is granted access to the data by *$*.

Strings

In YAQL, strings can be enclosed in " and '. Both types are absolutely equal and support all standard escape symbols including unicode code-points. In YAQL, both types of quotes are useful when you need to include one type of quotes into the other. In addition, ` is used to create a string where only one escape symbol ` is possible. This is especially suitable for regexp expressions.

If a string does not start with a digit or __ and contains only digits, _, and English letters, it is called identifier and can be used without quotes at all. An identifier can be used as a name for function, parameter or property in *$obj.property* case.

Functions

A function call has syntax of *functionName(functionParameters)*. Brackets are necessary even if there are no parameters. In YAQL, there are two types of parameters:

- **Positional parameters** `foo(1, 2, someValue)`
- **Named parameters** `foo(paramName1 => value1, paramName2 => 123)`

Also, a function can be called using both positional and named parameters: `foo(1, false, param => null)`. In this case, named arguments must be written after positional arguments. In `name => value`, *name* must be a valid identifier and must match the name of parameter in function definition. Usually, arguments can be passed in both ways, but named-only parameters are supported in YAQL since Python 3 supports them.

Parameters can have default values. Named parameters is a good way to pass only needed parameters and skip arguments which can be use default values, also you can simply skip parameters in function call: `foo(1,,3)`.

In YAQL, there are three types of functions:

- Regular functions: `max(1,2)`
- **Method-like functions, which are called by specifying an object for which the** function is called, followed by a dot and a function call: `stringValue.toUpper()`
- Extension methods, which can be called both ways: `len(string)`, `string.len()`

YAQL standard library contains hundreds of functions which belong to one of these types. Moreover, applications can add new functions and override functions from the standard library.

Operators

YAQL supports the following types of operators out of the box:

- Arithmetic: +. -, *, */*, *mod*
- Logical: =, *!*=, >=, <=, *and*, *or*, *not*
- Regexp operations: =~, *!*~
- Method call, call to the attribute: ., *?*.
- Context pass: ->
- Indexing: *[ ]*
- Membership test operations: *in*

Data structures

YAQL supports these types out of the box:

- Scalars

  YAQL supports such types as string, int. boolean. Datetime and timespan will be available after yaql 1.1 release.

- Lists

  List creation: `[1, 2, value, true]` Alternative syntax: `list(1, 2, value, true)` List elemenets can be accesessed by index: `$list[0]`

- Dictionaries

Dict creation: `{key1 => value1, true => 1, 0 => false}` Alternative syntax: `dict(key1 => value1, true => 1, 0 => false)` Dictionaries can be indexed by keys: `$dict[key]`. Exception will be raised if the key is missing in the dictionary. Also, you can specify value which will be returned if the key is not in the dictionary: `dict.get(key, default)`.

Note: During iteration through the dictionary, *key* can be called like: `$.key`

- (Optional) Sets

  Set creation: `set(1, 2, value, true)`

Note: YAQL is designed to keep input data unchanged. All the functions that look as if they change data, actually return an updated copy and keep the original data unchanged. This is one reason why YAQL is thread-safe.

Basic YAQL query operations

It is obvious that we can compare YAQL with SQL as they both are designed to solve similar tasks. Here we will take a look at the YAQL functions which have a direct equivalent with SQL.

We will use YAML from *HowTo: use YAQL in Python* as a data source in our examples.

Filtering

Note: Analog is SQL WHERE

The most common query to the data sets is filtering. This is a type of query which will return only elements for which the filtering query is true. In YAQL, we use `where` to apply filtering queries.

```
yaql> $.customers.where($.name = John)
```

```
- customer_id: 1
  name: John
  orders:
    - order_id: 1
      item: Guitar
      quantity: 1
```

Ordering

Note: Analog is SQL ORDER BY

It may be required to sort the data returned by some YAQL query. The `orderBy` clause will cause the elements in the returned sequence to be sorted according to the default comparer for the type being sorted. For example, the following query can be extended to sort the results based on the profession property.

```
yaql> $.customers.orderBy($.name)
```

```
- customer_id: 3
  name: Diana
  orders:
    - order_id: 4
      item: Drums
      quantity: 1
- customer_id: 1
  name: John
  orders:
    - order_id: 1
      item: Guitar
      quantity: 1
- customer_id: 2
  name: Paul
  orders:
    - order_id: 2
      item: Banjo
      quantity: 2
    - order_id: 3
      item: Piano
      quantity: 1
```

Grouping

Note: Analog is SQL GROUP BY

The `groupBy` clause allows you to group the results according to the key you specified. Thus, it is possible to group example json by gender.

```
yaql> $.customers.groupBy($.name)
```

```
- Diana:
  - customer_id: 3
    name: Diana
    orders:
      - order_id: 4
        item: Drums
        quantity: 1
- Paul:
  - customer_id: 2
    name: Paul
    orders:
      - order_id: 2
        item: Banjo
        quantity: 2
      - order_id: 3
        item: Piano
        quantity: 1
- John:
  - customer_id: 1
    name: John
    orders:
```

```
      - order_id: 1
        item: Guitar
        quantity: 1
```

So, here you can see the difference between `groupBy` and `orderBy`. We use the same parameter *name* for both operations, but in the output for `groupBy` *name* is located in additional place before everything else.

Selecting

Note: Analog is SQL SELECT

The `select` method allows building new objects out of objects of some collection. In the following example, the result will contain a list of name/orders pairs.

```
yaql> $.customers.select([$.name, $.orders])
```

```
- John:
  - order_id: 1
    item: Guitar
    quantity: 1
- Paul:
  - order_id: 2
    item: Banjo
    quantity: 2
  - order_id: 3
    item: Piano
    quantity: 1
- Diana:
  - order_id: 4
    item: Drums
    quantity: 1
```

Joining

Note: Analog is SQL JOIN

The `join` method creates a new collection by joining two other collections by some condition.

```
yaql> $.customers.join($.customers_city, $1.customer_id = $2.customer_id, {customer=>
↪$1.name, city=>$2.city, orders=>$1.orders})
```

```
- customer: John
  city: New York
  orders:
    - order_id: 1
      item: Guitar
      quantity: 1
- customer: Paul
  city: Saint Louis
  orders:
```

```
    - order_id: 2
      item: Banjo
      quantity: 2
    - order_id: 3
      item: Piano
      quantity: 1
- customer: Diana
  city: Mountain View
  orders:
    - order_id: 4
      item: Drums
      quantity: 1
```

Take an element from collection

YAQL supports two general methods that can help you to take elements from collection `skip` and `take`.

```
yaql> $.customers.skip(1).take(2)
```

```
- customer_id: 2
  name: Paul
  orders:
    - order_id: 2
      item: Banjo
      quantity: 2
    - order_id: 3
      item: Piano
      quantity: 1
- customer_id: 3
  name: Diana
  orders:
    - order_id: 4
      item: Drums
      quantity: 1
```

First element of collection

The `first` method will return the first element of a collection.

```
yaql> $.customers.first()
```

```
- customer_id: 1
  name: John
  orders:
    - order_id: 1
      item: Guitar
      quantity: 1
```

CHAPTER 5

Usage

This section is not ready yet.

Embedding YAQL

REPL utility

CHAPTER 6

Language reference

YAQL is a single expression language and as such does not have any block constructs, line formatting, end of statement marks or comments. The expression can be of any length. All whitespace characters (including newline) that are not enclosed in quote marks are stripped. Thus, the expressions may span multiple lines.

Expressions consist of:

- Literals
- Keywords
- Variable access
- Function calls
- Binary and unary operators
- List expressions
- Dictionary expressions
- Index expressions
- Delegate expressions

Terminology

- *YAQL* - the name of the language - acronym for *Yet Another Query Language*
- *yaql* - Python implementation of the YAQL language (this package)
- *expression* - a YAQL query that takes context as an input and produces result value
- *context* - an object that (directly or indirectly) holds all the data available to expression and all the function implementations accessible to expression
- *host* - the application that hosts the yaql interpreter. The host uses yaql to evaluate expressions, provides initial data, and decides which functions are going to be available to the expression. The host has ultimate power to

customize yaql - provide additional functions, operators, decide not to use standard library or use only parts of it, override function and operator behavior

- *variable* - any data item that is available through the context
- *function* - a Python callable that is exposed to the YAQL expression and can be called either explicitly or implicitly
- *delegate* - a Python callable that is available as a context variable (in expression data rather than registered in context)
- *operator* - a form of implicit function on one (unary operator) or two (binary operator) operands
- *alphanumeric* - consists of latin letters and digits (*A-Z*, *a-z*, *0-9*)

Literals

Literals refer to fixed values in expressions. YAQL has the following literals:

- Integer literals: `123`
- Floating point literals: `1.23`, `1.0`
- Boolean and null literals represented by *keywords* (see below)
- String literals enclosed in either single (‘) or double (”) quotes: `"abc"`, `'def'`. The backslash () character is used to escape characters that otherwise have a special meaning, such as newline, backslash itself, or the quote character
- Verbatim strings enclosed in back quote characters, for example `` `abc` ``, are used to suppress escape sequences. This is equivalent to `r'strings'` in Python and is especially useful for regular expressions

Keywords

Keyword is a sequence of characters that conforms to the following criteria:

- Consists of non-zero alphanumeric characters and an underscore (_)
- Doesn’t start with a digit
- Doesn’t start with two underscore characters (__)
- Is not enclosed in quote marks of any type

YAQL has only three predefined keywords: *true*, *false*, and *null* that have the value of similar JSON keywords.

There are also four keyword operators: *and*, *or*, *not*, *in*. However, this list is not fixed. The yaql host may decide to have additional keyword operators or not to have any of the four aforementioned keywords.

All other keywords have the value of their string representation. Thus, except for the predefined keywords and operators they can be considered as string literals and can be used anywhere where string is expected. However the opposite is not true. That is, keywords can be used as string literals but string literals cannot be used where a token is expected.

Examples:

- `John + Snow` - the same as `"John" + "Snow"`
- `true + love` - syntactically valid, but cannot be evaluated because there is no plus operator that accepts boolean and string (unless you define one)
- `not true` - evaluates to *false*, *not* is an operator

- `"foo"()` - invalid expression because the function name must be a token
- `John Snow` - invalid expression - two tokens with no operator between them

Variable access

Each YAQL expression is a function that takes inputs (arguments) and produces the result value (usually by doing some computations on those inputs). Expressions get the input through a *context* - an object that holds all the data and a list of functions, available for expression.

Besides the argument values, expressions may populate additional data items to the context. All these data are collectively known as a *variables* and available to all parts of an expression (unless overwritten with another value).

The syntax for accessing variable values is `$variableName` where *variableName* is the name of the variable. Variable names may consist of alphanumeric and underscore characters only. Unlike tokens, variable names may start with digit, any number of underscores and even be an empty string. By convention, the first (usually the single) function parameter is accessible through `$` expression (i.e. empty string variable name) which is an alias for `$1`. The usual case is to pass the main expression data in a single structure (document) and access it through the `$` variable.

If the variable with given name is not provided, it is assumed to be *null*. There is no built-in syntax to check if a variable exists to distinguish cases where it does not and when it is just set to null. However in the future such a function might be added to yaql standard library.

When the yaql parser encounters the `$variable` expression, it automatically translates it to the `#get_context_data("$variable")` function call. By default, the *#get_context_data* function returns a variable value from the current context. However the yaql host may decide to override it and provide another behavior. For example, the host may try to look up the value in an external data source (database) or throw an exception due to a missing variable.

Function calls

The power of YAQL comes from the fact that almost everything in YAQL is a function call (explicit or implicit) and any function may be overridden by the host. In YAQL there are two types of functions:

- *explicit function* - those that can be called from expressions
- *implicit (system) functions* - functions with predefined names that get called upon some operations. For example, `2 + 3` is translated to `#operator_+(2, 3)`. In this case, *#operator_+* is the name of the implicit function. However, because `#operator_+(2, 3)` is not a valid YAQL expression (because of *#*), implicit functions cannot be called explicitly but still can be redefined by the host.

The syntax for explicit function is:

```
call                  ::=   funcName ``('' [parameters] '')''
funcName              ::=   token
parameters            ::=   positionalParameters |
                            keywordParameters |
                            positionalParameters '','' keywordParameters
positionalParameters  ::=   parameter ('','' parameter)*
parameter             ::=   expression | empty-string
keywordParameters     ::=   keywordParameter ('','' keywordParameter)
keywordParameter      ::=   parameterName ``=>'' expression
parameterName         ::=   token
```

In simple words:

- The function name must be a token.
- Parameters may be positional, keyword or both. But keyword parameters may not come before positional.
- Positional parameters can be skipped if they have a default value, for example, `foo(1,,3)`.
- Keyword arguments must have a token name that must match the parameter name in the function declaration. Therefore, you must know the function signature for the right name.

Examples:

- `foo(2 + 3)`
- `bar(hello, world)`
- `baz(a,b, kwparam1 => c, kwparam2 => d)`

Functions have ultimate control over how they can be called. In particular:

- Each parameter may (and usually does) have an associated type check. That is, the function may specify that the expected parameter type and if it can be null.
- Usually, any parameters can be passed either by positional or keyword syntax. However, function declaration may force one particular way and make it positional-only or keyword-only.
- A function may have a variable number of positional (aka **args*) and/or keyword (aka ***kwarg*) arguments.
- In most languages, function arguments are evaluated prior to function invocation. This is not always true in YAQL. In YAQL, a function may declare a lazy argument. In this case, it is not evaluated and the function implementation receives a passed value as a callable or even as an AST, depending on how the parameter was declared. Thus in YAQL there is no special syntax for lambdas. `foo($ + 1)` may mean either "call *foo* with value of `$ + 1`" or "call *foo* with expression `$ + 1` as a parameter". In the latter case it corresponds to `foo(lambda *args, **kwargs: args[0] + 1)` in Python. Actual argument interpretation depends on the parameter declaration.
- Function may decide to disable keyword argument syntax altogether. For such functions, the `name => expr` expression will be interpreted as a positional parameter `yaql.language.utils.MappingRule(name, expr)` and the left side of `=>` can be any expression and not just a keyword. This allows for functions like `switch($ > 0 => 1, $ < 0 => -1, $ = 0 => 0)`.

Additionally, there are three subtypes of explicit functions. Suppose that there is a declared function `foo(string, int)`. By default, the syntax to call it will be `foo(something, 123)`. But it can be declared as a *method*. In this case, the syntax is going to be `something.foo(123)`. Because of the type checking, `something.foo(123)` will work since *something* is a string, but not the `123.foo(456)`. Thus *foo* becomes a method of a string type.

A function may also be declared as being an extension method. If foo were to be declared as an extension method it could be called both as a function (`foo(string, int)`) and as a method (`something.foo(123)`).

YAQL makes use of a full function signature to determine which function implementation needs to be executed. This allows several overloads of the same function as long as they differ by parameter count or parameter type, or anything else that allows unambiguous identification of the right overload from the function call expression. For example, `something.foo(123)` may be resolved to a completely different implementation of *foo* from that in `foo(something, 123)` if there are two functions with the name *foo* present in the context, but one of them was declared as a function while the other as a method. If several overloads are equally suitable for the call expression, an *AmbiguousFunctionException* or *AmbiguousMethodException* exception gets raised.

Operators

YAQL has both binary and unary operators, like most other languages do. Parentheses and => sequence are not considered as operators and handled internally by the yaql parser. However, it is possible to configure yaql to use sequence other than => for that purpose.

The list of available operators is not fixed and can be modified by the host. The following operators are available by default:

Binary operators:

Group	Operators
math operators	+, -, *, /, *mod*
comparision operators	>, <, >=, <=, =, *!=*
logical operators	*and*, *or*
method/member access	., *?.*
regex operators	=~, *!~*
membership operator	*in*
context passing operator	->

Unary operators:

Group	Operators
math operators	+, -
logical operators	*not*

YAQL supports for both prefix and suffix unary operators. However, only the prefix operators are provided by default.

In YAQL there are no built-in operators. The parser is given a list of all possible operator names (symbols), their associativity, precedence, and type, but it knows nothing about what operators are applicable for what operands. Each time a parser recognizes the `X OP Y` construct and *OP* is a known binary operator name, it translates the expression to `#operator_OP(X, Y)`. Thus. `2 + 3` becomes `#operator_+(2, 3)` where *#operator_+* is an implicit function with several implementations including the one for number addition and defined in standard library. The host may override it and even completely disable it. For unary operators, `OP X` (or `X OP` for suffix unary operators) becomes `#unary_operator_OP(X)`.

Upon yaql parser initialization, an operator might be given an alias name. In such cases, `X OP Y` is translated to `*ALIAS(X, Y)` and `OP X` to `*ALIAS(X)`. This decouples the operator implementation from the operator symbol. For example, the = operator has the *equal* alias. The host may configure yaql to have the == operator instead of = keeping the same alias so that operator implementation and all its consumers work equally well for the new operator symbol. In default configuration only = and *!=* operators have alias names.

For information on default operators, see the YAQL standard library reference.

List expressions

List expressions have the following form:

```
listExpression  ::=   ``['' [expressions] ``]''
expressions     ::=   expression (','' expression)*
```

When a yaql parser encounters an expression of the form `[A, B, C]`, it translates it into `#list(A, B, C)` (for arbitrary number of arguments).

Default *#list* function implementation in standard library produces a list (tuple) comprised of given elements. However,

the host might decide to give it a different implementation.

Map expressions

Map expressions have the following form:

```
mapExpression  ::=   ``{'' [mappings] ``}''
mappings       ::=   mapping (','' mapping)*
mapping        ::=   expression ``=>'' expression
```

When a yaql parser encounters an expression of the form `{A => X, B => Y}`, it translates it into `#map(A => X, B => Y)`.

The default *#map* implementation disables the keyword arguments syntax and thus receives a variable length list of mappings, which allows dictionary keys to be expressions rather than a keyword. It returns a (frozen) dictionary that itself can be used as a key in another map expression. For example, `{{a => b} => {[2 + 2, 2 * 2] => 4}}` is a valid YAQL expression though yaql REPL utility will fail to display its output due to the fact that it is not JSON-compatible.

Index expressions

Index expressions have the following form:

```
indexExpression  ::=   expression listExpression
```

Examples:

- `[1, 2, 3][0]`
- `$arr[$index + 1]`
- `{foo => 1, bar => 2}[foo]`

When a yaql parser encounters such an expression, it translates it into `#indexer(expression, index)`.

The standard library provides a number of *#indexer* implementations for different types.

The right side of the index expression is a list expression. Therefore, an expression like `$foo[1, x, null]` is also a valid YAQL expression and will be translated to `#indexer($foo, 1, x, null)`. However, any attempt to evaluate such expression will result in *NoMatchingFunctionException* exception because there is no *#indexer* implementation that accepts such arguments (unless the host defines one).

Delegate expressions

Delegate expressions is an optional language feature that is disabled by default. It makes possible to pass delegates (callables) as part of the context data and invoke them from the expression. It has the same syntax as explicit function calls with the only difference being that instead of function name (keyword) there is a non-keyword expression that must produce the delegate.

Examples:

- `$foo(1, arg => 2)` - call delegate returned by `$foo` with parameters `(1, arg => 2)`
- `[$foo, $bar][0](x)` - the same as `$foo(x)`
- `foo()()` - can be written as `(foo())()` - `foo()` must return a delegate

Delegate expressions are translated into `#call(callable, arguments)`. Thus `$foo(1, 2)` becomes `#call($foo, 1, 2)`.

The default implementation of `#call` invokes the result of the evaluation of its first arguments with the given arguments.

CHAPTER 7

Customizing and extending yaql

Configuring yaql parser

yaql has two main points of customization:

- yaql engine settings allow one to configure the query language and execution flags shared by all queries that are processed by the same YAQL parser. This includes the list of available operators, yaql resources quotas, and other engine parameters.
- By customizing the yaql context object, one can change the list of available functions (add new, override existing) and change naming conventions.

Engine options are supplied to the *yaql.language.factory.YaqlFactory* class. YaqlFactory is used to create instances of the YaqlEngine, that is the YAQL parser. This is done by calling the *create* method of the factory. Once the engine is created, it captures all the factory options so that they cannot be changed for that particular parser any longer. In general, it is recommended to have one yal engine instance per application, because construction of the parser is an expensive operation and the parser has no internal state and thus can be reused for several queries, including in different threads. However, the host may have several YAQL parsers for different option sets or dialects.

On the contrary, the context object is cheap to create and is mutable by design, since it holds the input data for the query. In most cases it is a good idea to execute each query in its own context, although all such contexts might be the children of some other, fixed context that is created just once.

Customizing operators

YaqlFactory object holds an operator table that is recognized by the parser produced by it. By default, it is prepopulated with standard operators and most applications never need to do anything here. However, if the host wants to have some custom operator symbol available in its expressions, this table needs to be modified. *YaqlFactory* holds the operator symbols and other information about the operator that is relevant to the parser, but not the implementations. The implementations (what operators actually do) are put in the *context* and can be configured for each expression, but the list of available operator symbols cannot be changed for the parser once it has been built.

Each operator in the table is represented by the tuple ‘(op_symbols, op_type, op_alias):

- op_symbols are the operator symbols. There are no limitations on how the operators can be called as long as they do not contain whitespaces. It can be one symbol (like +), several symbols (like =~) or even a word (like *not*). List/index and dictionary expressions require *[]* and *{}* binary left associative operators to be present in the table. Otherwise corresponding constructions will not work (and can be disabled by removing corresponding operators from the table)
- op_type is one of the values in *yaql.language.factory.OperatorType* enumeration: BINARY_LEFT_ASSOCIATIVE and BINARY_RIGHT_ASSOCIATIVE for binary operators, PREFIX_UNARY and SUFFIX_UNARY for unary operators, NAME_VALUE_PAIR for the keyword/mapping pseudo-operator (that is =>, by default).
- op_alias is the alias name for the operator. See YAQL language reference on how operator aliases are used. Aliases are optional and most operators do not have it and thus are represented by a tuple of two elements.

Operators are grouped by their precedence. Operators with a higher precedence come first in the operator table. Operators within the same group have the same precedence. Groups are separated by an empty tuple (*()*).

The operator table, which is a list of tuples, is available through the *operators* attribute of the factory and is open for modification. To simplify the editing, *YaqlFactory* provides the *insert_operator* helper method to insert an operator before of after some other existing operator to get the desired precedence.

Execution options

Execution options are the settings and flags that affect execution of each query and are accessible and processed by both yaql runtime and standard library functions.

Options are passed to the *create* method of the *YaqlFactory* class in a plain key-value dictionary. The factory does not process the dictionary but rather attaches the options to the constructed engine (YAQL parser) after which they cannot be changed. However, the engine provides a *copy* method that can be used to clone the engine with different execution options.

The options that are honored by the yaql are:

- *"yaql.limitIterators": <INT>* limit iterators by the given number of elements. When set, each time any function declares its parameter to be iterator, that iterator is modified to not produce more than a given number of items. Also, upon the expression evaluation, all the output collections and iterators are limited as well. If not set (or set to -1) the result data is allowed to contain endless iterators that would cause errors if the result where to be serialized (to JSON or any other format). Default is -1 (do not limit).
- *"yaql.memoryQuota": <INT>* - the memory usage quota (in bytes) for all data produced by the expression (or any part of it). Default is -1 (do not limit).
- *"yaql.convertTuplesToLists": <True|False>*. When set to true, yaql converts all tuples in the expression result to lists. The default is *True*.
- *"yaql.convertSetsToLists": <True|False>*. When set to true, yaql converts all sets in the expression result to lists. Otherwise the produced result may contain sets that are not JSON-serializable. The default is *False*.
- *"yaql.iterableDicts": <True|False>*. When set to true, dictionaries are considered to be iterable and iteration over dictionaries produces their keys (as in Python and yaql 0.2). Defaults to *False*.

Consumers are free to use their own settings or use the options dictionary to provide some other environment information to their own custom functions.

Other engine customizations

YaqlFactory class initializer has two optional parameters that can be used to further customize the YAQL parser:

- *keyword_operator* allows one to configure keyword/mapping symbol. The default is =>. Ability to pass named arguments can be disabled altogether if *None* or empty string is provided.
- *allow_delegates* enables or disables delegate expression parsing. Default is False (disabled).

Working with contexts

Context is an interface that yaql runtime uses to obtain a list of available functions and variables. Any context object must implement *yaql.language.contexts.ContextBase* interface and yaql provides several such implementations ranging from the *yaql.language.contexts.Context* class, that is a basic context implementation, to contexts that allow one to merge several other contexts into one or link an existing context into the list of contexts.

Any context may have a parent context. Any lookup that is done in the context is also performed in its parent context, extending all the way up its chain of contexts. During expression evaluation, yaql can create a long chain of contexts that are all children of the context that was originally passed with the query.

Most of the yaql customizations are achieved by context manipulations. This includes:

- Overriding YAQL functions
- Building context chains and evaluating sub-expressions in different contexts
- Composing context chains from pre-built contexts
- Having custom *ContextBase* implementations and mixing them with regular contexts in the single chain

In fact, it is the context which provides the entry point for expression evaluation. And thus custom context implementations may completely change the way queries are evaluated.

There are three ways to create a context instance:

1. Directly instantiate one of *ContextBase* implementations to get an empty context
2. Call *create_child_context* method on any existing context object to get a child context

#. Use *yaql.create_context* function to creates the root context that is prepopulated with YAQL standard library functions

yaql.create_context allows one to selectively disable standard library modules.

Naming conventions

Naming conventions define how Python functions and parameter names are translated into YAQL names. Conventions are implementations of the *yaql.language.conventions.Convention* interface that has just two methods: one to translate the function name and another to translate the function parameter name.

yaql has two implementations included:

- *yaql.language.conventions.CamelCaseConvention' that translates Python conventions into camel case. For example, it will convert 'my_func(arg_name)* into *myFunc(argName)*. This convention is used by default.
- 'yaql.language.conventions.PythonConvention' that leaves function and parameter names intact.

Each context, either directly or indirectly through its parent context, is configured to use some convention. When a function is registered in the context, its name and parameters are translated with the convention methods. Also, regardless of convention used, all trailing underscores are stripped from the names. This makes it possible to define several Python functions that differ only by trailing underscores and get the same name in YAQL (to create several overloads of single function). Also, this allow one to have function or parameter names that would otherwise conflict with Python keywords.

Instance of convention class can be specified as a context initializer parameter or as a parameter of *yaql.create_context* function. Child contexts created with the *create_child_context* method inherit their parent convention.

Extending yaql

Extending yaql with new functions

For a function to become available to YAQL queries, it must be present in the provided context object. The default context implementation (*yaql.language.contexts.Context*) has a *register_function* method to register the function implementation.

In yaql, all functions are represented by instances of the *yaql.language.specs.FunctionDefinition* class. FunctionDefinition describes the complete function signature including:

- Function name
- List of parameters - instances of *yaql.language.specs.ParameterDefinition*
- Function payload (Python callable)
- Function type: function, method or extension method
- The flag to disable the keyword arguments syntax for the function
- Documentation string
- Custom function metadata (dict)

register_function method can accept either an instance of the *FunctionDefinition* class or a regular Python function. In the latter case, it constructs a *FunctionDefinition* instance from the declaration of the function using Python introspection. Because a YAQL function signature has much more information than the Python one, yaql provides a number of function decorators that can be used to fill the missing properties.

The decorators are located in the *yaql.language.specs* module. Below is the list of available function decorators:

- `@name(function_name)`: set function name to be *function_name* rather than its Python name
- `@parameter(...)` is used to declare the type of one of the function parameters
- `@inject(...)` is used to declare a hidden function parameter
- `@method` declares function to be YAQL method
- `@extension_method` declares function to be YAQL extension method
- `@no_kwargs` disables the keyword arguments syntax for the function
- `@meta(name, value)` appends the *name* attribute with the given value to the function metadata dictionary

Specifying function parameter types

When yaql constructs *FunctionDefinition*, it collects all possible information about its parameters. For each parameter, it records its name, position, whether it is a keyword-only argument (available in Python 3), whether it is an **args* or ***kwargs*, and its default parameter value.

The only parameter attribute that cannot be obtained through retrospection is the parameter type. For that purpose, yaql has a `@parameter(name, type)` decorator that can be used to explicitly declare the parameter type. *name* must match the name of one of the function parameters, and *type* must be of the *yaql.language.yaqltypes.SmartType* type.

SmartType is the base class for all yaql type descriptors - classes that check if the value is compatible with the desired type and can do type conversion between compatible types.

YAQL type system slightly differs from Python's:

- Strings are not considered to be collections of characters
- Booleans are not integers
- Dictionaries are not iterable
- For most of the types one can specify if the *null* (*None*) value is acceptable

yaql.language.yaqltypes module has many useful smart-type classes. The most generic smart-type for primitive types is the *PythonType* class, that validates if the value is instance of a given Python type. Due to the mentioned differences between YAQL and Python type systems and because Python types have a lot of nuances (several string types, differences between Python 2 and Python 3, separation between mutable and immutable type versions: list-tuple, set-frozenset, dict-FrozenDict, which is missing in Python and provided by the yaql instead), yaql provides specialized smart-types for most primitive types:

- *String* - str and unicode
- *Integer*
- *Number* - integer of float
- *DateTime*
- *Sequence* - fixed-size iterable collection, except for the dictionary
- *Iterable* - any iterable or generator
- *Iterator* - iterator over the iterable

And several specialized variants that enforce particular representation in the YAQL syntax:

- *Keyword*
- *BooleanConstant*
- *NumericConstant*
- *StringConstant*

It is also possible to aggregate several smart-types so that the value can be of any given type or conform to all of them:

- *AnyOf*
- *Chain*
- *NotOfType*

These three smart-types accept other smart-type(s) as their initializer parameter(s).

In addition to the smart-types, the second parameter of the *@parameter* can be a Python type. For example, `@parameter("name", unicode)` or `@parameter("name", unicode, nullable=True)`. In this case the Python type is automatically wrapped in the *PythonType* smart-type. If nullability is not specified, yaql tries to infer it from the parameter declaration - it is nullable only if the parameter has its default value set to *None*.

Lazy evaluated function parameters

All the smart-types from the previous section are for parameters that are evaluated before the function gets invoked. But sometimes the function might need the parameter to remain unevaluated so that it can be evaluated by the function itself, possibly with additional parameters or in a different context.

There are two possible representations of non-evaluated arguments:

- Get it as a Python callable that the function can call to do the evaluation
- Get it as a YAQL expression (AST), that can be analyzed

The first method is available through the *Lambda* smart-type. The parameter, which is declared as a `Lambda()`, has an **args/**kwargs* signature and can be called from the function: `parameter(arg1, arg2)`. If it was declared as `Lambda(with_context=True)` the function may invoke it in a context, other than that which is used for the function: `parameter(new_context, arg1, arg2)`. `Lambda(method=True)` specifies that the parameter must be a method and the caller can specify the receiver object for it: `parameter(receiver, arg1, arg2)`. Parameters can also be combined: `Lambda(with_context=True, method=True)` so the callable is invoked as `parameter(receiver, new_context, arg1, arg2)`. All supplied callable arguments are automatically published to the *$1* (*$*), *$2* and so on context variables for the context in which the callable will be executed.

The second method is available through the *YaqlExpression* smart-type. It also allows one to request the parameter to be of a particular expression type rather than an arbitrary YAQL expression.

Auto-injected function parameters

Besides regular parameters, yaql also supports auto-injected (hidden) parameters. This is also known as a function parameter dependency injection. The values of injected parameters come from the yaql runtime rather than from the caller. Functions use injected parameters to get information on their execution environment.

Auto-injected parameters are declared using the `@inject(...)` decorator, which has exactly the same signature as *@parameter* with the only difference being that *@inject* checks that that the supplied smart-type is an instance of the *yaql.language.yaqltypes.HiddenParameterType* class (in addition to *SmartType*), whereas the *@parameter* decorator checks that it is not. This difference exists to clearly distinguish explicitly passed parameters from those that are injected by the system.

yaql has the following hidden parameter smart types:

- *Context* - injects the current function context object
- *Engine* - injects *YaqlEngine* object that was used to parse the expression. Engine object may be used to access execution options or to parse some other expression
- *FunctionDefinition* - *FunctionDefinition* object of the function. May be used to obtain function metadata and doc-string
- *Delegate* - injects a Python callable to some other YAQL function by its name. This is a convenient way to call one YAQL function from another without depending on its Python implementation signature and location. The syntax is very similar to *Lambda* smart-type
- *Super* - similar to *Delegate* - injects callable to an overload of itself from the parent context. Useful when the function overload wants to call its base implementation (analogous to Python's `super()`)
- *Receiver* - injects a method receiver object if the function was called as a method and *None* otherwise. Can be used in an extension method to distinguish the case, when it was invoked as a method rather than as a function. Do not do it without a good reason!
- *YaqlInterface* - injects a convenient wrapper (*YaqlInterface*) around yaql functionality, which also encapsulates many of the values above

Auto-injected parameters may appear anywhere in the function signature as they do not affect caller syntax. Implementations can add additional hidden parameters without breaking existing queries. However, it is important to call YAQL function implementations through the yaql mechanisms (such as *Delegate*), rather than to call their Python implementations directly.

Automatic parameters

In some cases there is no need to declare the parameter at all. yaql uses parameter name and default value to guess the parameter type if it was not declared.

If the parameter name is *context* or *__context* it will automatically be treated as if it was declared as a *Context*. *engine*/*__engine* is considered as an *Engine*, and *yaql_interface*/*__yaql_interface* is considered as a *YaqlInterface*.

The host can override this logic by providing a callable to Context's *register_function* method through the *parameter_type_func* parameter. When yaql encounters an undeclared parameter, it calls this function, passing the parameter name as an argument, and expects it to return a smart-type for the parameter.

If the *parameter_type_func* callable returned *None*, yaql would assume that the smart type should be *PythonType(object)*, that is anything, except for the *None* value, unless the parameter had the default value *None*.

Function resolution rules

Function resolution rules are used to determine the correct overload of the function when more than one overload is present in the context. Each time a function with a given list of parameters is called yaql does the following:

1. Walks through the chain of context objects and collects all the implementations with a given name and appropriate type (either functions and extension methods or methods and extensions methods, depending on the call syntax).
2. All found overloads are organized into layers so that overloads from the same context will be put in the same layer whereas overloads from different contexts are in different layers. Overloads from contexts that are closer to the initial context have precedence over those which were obtained from the parent contexts. Also *FunctionDefinition* may have a flag that prevents all overload lookups in the parent contexts. If the search encounters an overload with such a flag, it does not go any further in the chain.
3. Scan all found overloads and exclude those, that cannot be called by the given syntax. This can happen because the overload has more mandatory parameters than the arguments in the calling expression, or because it passes the argument using the keyword name and no such parameter exists.
4. Validates laziness of overload parameters. If at least one function overload has a lazy evaluated parameter all other overloads must have it in the same position. Violation of this rule causes an exception to be thrown.
5. All the non-lazy parameters are evaluated. The result values are validated by appropriate smart-type instances corresponding to each parameter of each overload. All the overloads that are not type-compatible with the given arguments are excluded in each layer.
6. Take first non-empty layer. If no such layer exists (that is all the overloads were excluded) then throw an exception.
7. If the found layer has more than one overload, then we have an ambiguity. In this case an exception is thrown since we cannot unambiguously determine the right overload.
8. Otherwise, call the single overload with previously evaluated arguments.

Function development hints

- Avoid side effects in your functions, unless you absolutely have to.
- Do not make changes to the data structures coming from the parameters or the context. Functions that modify the data should return the modified copy rather than touch the original.
- If you need to make changes to the context, create a child context and make them there. It is usually possible to pass the new context to other parts of the query.

- Strongly prefer immutable data structures over mutable ones. Use *tuple's rather than 'list's, 'frozenset* instead of *set*. Python does not have a built-in immutable dictionary class so yaql provides one on its own - *yaql.language.utils.FrozenDict*.
- Do not call Python implementation of YAQL functions directly. yaql provides plenty of ways to do so.
- Do not reuse contexts between multiple queries unless it is intentional. However all of these contexts can be children of a single prepared context.
- Do not register all the custom functions for each query. It is better to prepare all the contexts with functions at the beginning and then use child contexts for each query executed.

CHAPTER 8

Contributing

If you would like to contribute to the development of OpenStack, you must follow the steps in this page:

http://docs.openstack.org/infra/manual/developers.html

Once those steps have been completed, changes to OpenStack should be submitted for review via the Gerrit tool, following the workflow documented at:

http://docs.openstack.org/infra/manual/developers.html#development-workflow

Pull requests submitted through GitHub will be ignored.

Bugs should be filed on Launchpad, not GitHub:

https://bugs.launchpad.net/yaql