python-sqlparse Documentation

Release 0.1.3

Andi Albrecht

July 30, 2011

CONTENTS

CHAPTER

ONE

INTRODUCTION

sqlparse is a non-validating SQL parser for Python. It provides support for parsing, splitting and formatting SQL statements. The module is released under the terms of the New BSD license.

Visit the project page at http://python-sqlparse.googlecode.com for further information about this project.

1.1 Download & Installation

The latest released version can be obtained from the downloads page on the project's website. To extract the source archive and to install the module on your system run

```
$ tar cvfz python-sqlparse-VERSION.tar.gz
$ cd python-sqlparse/
$ sudo python setup.py install
```

Alternatively you can install sqlparse from the Python Packge Index with your favorite tool for installing Python modules. For example when using pip run **pip install sqlparse**.

1.2 Getting Started

The sqlparse module provides three simple functions on module level to achieve some common tasks when working with SQL statements. This section shows some simple usage examples of these functions.

Let's get started with splitting a string containing one or more SQL statements into a list of single statements using split():

```
>>> import sqlparse
>>> sql = 'select * from foo; select * from bar;'
>>> sqlparse.split(sql)
[u'select * from foo; ', u'select * from bar;']
```

The end of a statement is identified by the occurrence of a semicolon. Semicolons within certain SQL constructs like BEGIN ... END blocks are handled correctly by the splitting mechanism.

SQL statements can be beautified by using the format() function.

```
>>> sql = 'select * from foo where id in (select id from bar);'
>>> print sqlparse.format(sql, reindent=True, keyword_case='upper')
SELECT *
FROM foo
WHERE id IN
```

```
  (SELECT id
   FROM bar);
```

In this case all keywords in the given SQL are uppercased and the indentation is changed to make it more readable. Read *Formatting of SQL Statements* for a full reference of supported options given as keyword arguments to that function.

Before proceeding with a closer look at the internal representation of SQL statements, you should be aware that this SQL parser is intentionally non-validating. It assumes that the given input is at least some kind of SQL and then it tries to analyze as much as possible without making too much assumptions about the concrete dialect or the actual statement. At least it's up to the user of this API to interpret the results right.

When using the `parse()` function a tuple of `Statement` instances is returned:

```
>>> sql = 'select * from "someschema"."mytable" where id = 1'
>>> parsed = sqlparse.parse(sql)
>>> parsed
(<Statement 'select...' at 0x9ad08ec>,)
```

Each item of the tuple is a single statement as identified by the above mentioned `split()` function. So let's grab the only element from that list and have a look at the `tokens` attribute. Sub-tokens are stored in this attribute.

```
>>> stmt = parsed[0]  # grab the Statement object
>>> stmt.tokens
(<DML 'select' at 0x9b63c34>,
 <Whitespace ' ' at 0x9b63e8c>,
 <Operator '*' at 0x9b63e64>,
 <Whitespace ' ' at 0x9b63c5c>,
 <Keyword 'from' at 0x9b63c84>,
 <Whitespace ' ' at 0x9b63cd4>,
 <Identifier '"somes...' at 0x9b5c62c>,
 <Whitespace ' ' at 0x9b63f04>,
 <Where 'where ...' at 0x9b5caac>)
```

Each object can be converted back to a string at any time:

```
>>> stmt.to_unicode()
u'select * from "someschema"."mytable" where id = 1'
>>> stmt.tokens[-1].to_unicode()  # or just the WHERE part
u'where id = 1'
```

Details of the returned objects are described in *Analyzing the Parsed Statement*.

1.3 Development & Contributing

To check out the latest sources of this module run

```
$ hg clone http://python-sqlparse.googlecode.com/hg/ python-sqlparse
```

to check out the latest sources from the Mercurial repository.

Please file bug reports and feature requests on the project site at http://code.google.com/p/python-sqlparse/issues/entry or if you have code to contribute upload it to http://codereview.appspot.com and add albrecht.andi@googlemail.com as reviewer.

For more information about the review tool and how to use it visit it's project page: http://code.google.com/p/rietveld.

CHAPTER

TWO

SQLPARSE – PARSE SQL STATEMENTS

The `sqlparse` module provides the following functions on module-level.

`sqlparse.`**`split`**`(`*`sql`*`)`
Split *sql* into single statements.

Returns a list of strings.

`sqlparse.`**`format`**`(`*`sql`*`, `*`**options`*`)`
Format *sql* according to *options*.

Available options are documented in *Formatting of SQL Statements*.

Returns the formatted SQL statement as string.

`sqlparse.`**`parse`**`(`*`sql`*`)`
Parse sql and return a list of statements.

sql is a single string containting one or more SQL statements.

Returns a tuple of `Statement` instances.

2.1 Formatting of SQL Statements

The `format()` function accepts the following keyword arguments.

`keyword_case` Changes how keywords are formatted. Allowed values are "upper", "lower" and "capitalize".

`identifier_case` Changes how identifiers are formatted. Allowed values are "upper", "lower", and "capitalize".

`strip_comments` If `True` comments are removed from the statements.

`reindent` If `True` the indentations of the statements are changed.

`indent_tabs` If `True` tabs instead of spaces are used for indentation.

`indent_width` The width of the indentation, defaults to 2.

`output_format` If given the output is additionally formatted to be used as a variable in a programming language. Allowed values are "python" and "php".

CHAPTER

THREE

ANALYZING THE PARSED STATEMENT

When the `parse()` function is called the returned value is a tree-ish representation of the analyzed statements. The returned objects can be used by applications to retrieve further information about the parsed SQL.

3.1 Base Classes

All returned objects inherit from these base classes. The `Token` class represents a single token and `TokenList` class is a group of tokens. The latter provides methods for inspecting it's child tokens.

class `sqlparse.sql.`**Token**(*ttype*, *value*)
Base class for all other classes in this module.

It represents a single token and has two instance attributes: `value` is the unchange value of the token and `ttype` is the type of the token.

flatten()
Resolve subgroups.

has_ancestor(*other*)
Returns `True` if *other* is in this tokens ancestry.

is_child_of(*other*)
Returns `True` if this token is a direct child of *other*.

is_group()
Returns `True` if this object has children.

is_whitespace()
Return `True` if this token is a whitespace token.

match(*ttype*, *values*, *regex=False*)
Checks whether the token matches the given arguments.

ttype is a token type. If this token doesn't match the given token type. *values* is a list of possible values for this token. The values are OR'ed together so if only one of the values matches `True` is returned. Except for keyword tokens the comparison is case-sensitive. For convenience it's ok to pass in a single string. If *regex* is `True` (default is `False`) the given values are treated as regular expressions.

to_unicode()
Returns a unicode representation of this object.

within(*group_cls*)
Returns `True` if this token is within *group_cls*.

Use this method for example to check if an identifier is within a function: `t.within(sql.Function)`.

class `sqlparse.sql.`**`TokenList`**(*tokens=None*)
A group of tokens.

It has an additional instance attribute `tokens` which holds a list of child-tokens.

`flatten`()
Generator yielding ungrouped tokens.

This method is recursively called for all child tokens.

`group_tokens`(*grp_cls*, *tokens*, *ignore_ws=False*)
Replace tokens by an instance of *grp_cls*.

`insert_before`(*where*, *token*)
Inserts *token* before *where*.

`token_first`(*ignore_whitespace=True*)
Returns the first child token.

If *ignore_whitespace* is `True` (the default), whitespace tokens are ignored.

`token_index`(*token*)
Return list index of token.

`token_next`(*idx*, *skip_ws=True*)
Returns the next token relative to *idx*.

If *skip_ws* is `True` (the default) whitespace tokens are ignored. `None` is returned if there's no next token.

`token_next_by_instance`(*idx*, *clss*)
Returns the next token matching a class.

idx is where to start searching in the list of child tokens. *clss* is a list of classes the token should be an instance of.

If no matching token can be found `None` is returned.

`token_next_by_type`(*idx*, *ttypes*)
Returns next matching token by it's token type.

`token_next_match`(*idx*, *ttype*, *value*, *regex=False*)
Returns next token where it's `match` method returns `True`.

`token_prev`(*idx*, *skip_ws=True*)
Returns the previous token relative to *idx*.

If *skip_ws* is `True` (the default) whitespace tokens are ignored. `None` is returned if there's no previous token.

`tokens_between`(*start*, *end*, *exclude_end=False*)
Return all tokens between (and including) start and end.

If *exclude_end* is `True` (default is `False`) the end token is included too.

3.2 SQL Representing Classes

The following classes represent distinct parts of a SQL statement.

class `sqlparse.sql.`**`Statement`**(*tokens=None*)
Represents a SQL statement.

get_type()
Returns the type of a statement.

The returned value is a string holding an upper-cased reprint of the first DML or DDL keyword. If the first token in this group isn't a DML or DDL keyword "UNKNOWN" is returned.

class sqlparse.sql.**Comment**(*tokens=None*)
A comment.

class sqlparse.sql.**Identifier**(*tokens=None*)
Represents an identifier.

Identifiers may have aliases or typecasts.

get_alias()
Returns the alias for this identifier or `None`.

get_name()
Returns the name of this identifier.

This is either it's alias or it's real name. The returned valued can be considered as the name under which the object corresponding to this identifier is known within the current statement.

get_parent_name()
Return name of the parent object if any.

A parent object is identified by the first occuring dot.

get_real_name()
Returns the real name (object name) of this identifier.

get_typecast()
Returns the typecast or `None` of this object as a string.

has_alias()
Returns `True` if an alias is present.

is_wildcard()
Return `True` if this identifier contains a wildcard.

class sqlparse.sql.**IdentifierList**(*tokens=None*)
A list of `Identifier`'s.

get_identifiers()
Returns the identifiers.

Whitespaces and punctuations are not included in this list.

class sqlparse.sql.**Where**(*tokens=None*)
A WHERE clause.

class sqlparse.sql.**Case**(*tokens=None*)
A CASE statement with one or more WHEN and possibly an ELSE part.

get_cases()
Returns a list of 2-tuples (condition, value).

If an ELSE exists condition is None.

class sqlparse.sql.**Parenthesis**(*tokens=None*)
Tokens between parenthesis.

class sqlparse.sql.**If**(*tokens=None*)
An 'if' clause with possible 'else if' or 'else' parts.

class sqlparse.sql.**For**(*tokens=None*)
A 'FOR' loop.

class sqlparse.sql.**Assignment**(*tokens=None*)
An assignment like 'var := val;'

class sqlparse.sql.**Comparison**(*tokens=None*)
A comparison used for example in WHERE clauses.

CHAPTER

FOUR

USER INTERFACES

sqlformat The `sqlformat` command line script ist distributed with the module. Run **sqlformat –help** to list available options and for usage hints.

sqlformat.appspot.com An example Google App Engine application that exposes the formatting features using a web front-end. See http://sqlformat.appspot.com for details. The source for this application is available from a source code check out of the `sqlparse` module (see `extras/appengine`).

CHAPTER

FIVE

CHANGES IN PYTHON-SQLPARSE

5.1 Release 0.1.3 (Jul 29, 2011)

Bug Fixes

- Improve parsing of floats (thanks to Kris).
- When formatting a statement a space before LIMIT was removed (issue35).
- Fix strip_comments flag (issue38, reported by ooberm...@gmail.com).
- Avoid parsing names as keywords (issue39, reported by djo...@taket.org).
- Make sure identifier lists in subselects are grouped (issue40, reported by djo...@taket.org).
- Split statements with IF as functions correctly (issue33 and issue29, reported by charles....@unige.ch).
- Relax detection of keywords, esp. when used as function names (issue36, nyuhu...@gmail.com).
- Don't treat single characters as keywords (issue32).
- Improve parsing of stand-alone comments (issue26).
- Detection of placeholders in paramterized queries (issue22, reported by Glyph Lefkowitz).
- Add parsing of MS Access column names with braces (issue27, reported by frankz...@gmail.com).

Other

- Replace Django by Flask in App Engine frontend (issue11).

5.2 Release 0.1.2 (Nov 23, 2010)

Bug Fixes

- Fixed incorrect detection of keyword fragments embed in names (issue7, reported and initial patch by andyboyko).
- Stricter detection of identfier aliases (issue8, reported by estama).
- WHERE grouping consumed closing parenthesis (issue9, reported by estama).
- Fixed an issue with trailing whitespaces (reported by Kris).
- Better detection of escaped single quotes (issue13, reported by Martin Brochhaus, patch by bluemaro with test case by Dan Carley).
- Ignore identifier in double-quotes when changing cases (issue 21).

- Lots of minor fixes targeting encoding, indentation, statement parsing and more (issues 12, 14, 15, 16, 18, 19).
- Code cleanup with a pinch of refactoring.

5.3 Release 0.1.1 (May 6, 2009)

Bug Fixes

- Lexers preserves original line breaks (issue1).
- Improved identifier parsing: backtick quotes, wildcards, T-SQL variables prefixed with @.
- Improved parsing of identifier lists (issue2).
- Recursive recognition of AS (issue4) and CASE.
- Improved support for UPDATE statements.

Other

- Code cleanup and better test coverage.

5.4 Release 0.1.0 (Apr 8, 2009)

- Initial release.

CHAPTER

SIX

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

PYTHON MODULE INDEX