Package: sqlparseR (via r-universe)

November 3, 2024

Type Package

Title Wrapper for 'Python' Module 'sqlparse': Parse, Split, and Format 'SQL'

Version 0.1.0

Description Wrapper for the non-validating 'SQL' parser 'Python' module 'sqlparse' <https://github.com/andialbrecht/sqlparse>. It allows parsing, splitting, and formatting 'SQL' statements.

License GPL-3

Encoding UTF-8

LazyData true

Imports reticulate (>= 1.13)

NeedsCompilation no

Author Michael Simmler [aut, cre]

Maintainer Michael Simmler <michael.simmler@agroscope.admin.ch>

Date/Publication 2019-09-20 09:50:02 UTC

Repository https://agroscope-ch.r-universe.dev

RemoteUrl https://github.com/agroscope-ch/sqlparser

RemoteRef HEAD

RemoteSha a52adfaf156995dee3267a0bde1cd2b4be8a0884

Contents

`install_sqlparse_py` *Install sqlparse Python package*

Description

Install the *sqlparse* Python package into a virtual environment or conda environment.

Usage

```
install_sqlparse_py(method = "auto", conda = "auto",
                    envname = NULL, skip_if_available = FALSE)
```

Arguments

`method`	Installation method passed to `py_install`. Options: "auto", "virtualenv", and "conda". Default: "auto"
`conda`	Path to conda executable passed to `py_install`. Alternatively, "auto" to find conda using the PATH and other conventional install locations. Default: "auto"
`envname`	The name, or full path, of the environment in which the *sqlparse* Python package is to be installed. Alternatively, `NULL` to use the active environment as set by the `RETICULATE_PYTHON_ENV` variable or, if that is unset, the r-reticulate environment. Default: `NULL`
`skip_if_available`	Boolean; if `TRUE` the installation is skipped in case the *sqlparse* Python module can be found on the system (search not limited to `envname`). Default: `FALSE`

Value

0 on successful installation or 1 in case an error was raised

Examples

```
## Not run:
install_sqlparse_py()

## End(Not run)
```

sql_format *Format SQL Statements*

Description

Beautifies SQL statements according to numerous formatting settings.

Usage

```
sql_format(sql, keyword_case = NULL, identifier_case = NULL,
           strip_comments = TRUE, reindent = FALSE, indent_width = 2,
           indent_tabs = FALSE, indent_after_first = FALSE,
           indent_columns = FALSE, reindent_aligned = FALSE,
           use_space_around_operators = FALSE, wrap_after = NULL,
           comma_first = FALSE, truncate_strings = NULL,
           truncate_char = "[...]", encoding = NULL)
```

Arguments

sql	Character string containing one or more SQL statements to be formatted.
keyword_case	Character string specifying how keywords are formatted. Options: "upper", "lower"", and "capitalize". Default: NULL
identifier_case	Character string specifying how identifiers are formatted. Options: "upper", "lower"", and "capitalize". Default: NULL
strip_comments	Boolean; if TRUE comments are removed from the SQL statements. Default: TRUE
reindent	Boolean; if TRUE the indentations of the statements are changed. Default: FALSE
indent_width	Positive integer specifying the width of the indentation. Default: 2
indent_tabs	Boolean; if TRUE tabs instead of spaces are used for indentation. Default: FALSE
indent_after_first	Boolean; if TRUE second line of statement is indented (e.g. after SELECT). Default: FALSE
indent_columns	Boolean; if TRUE all columns are indented by indent_width instead of keyword length. Default: FALSE
reindent_aligned	Boolean; if TRUE the statements are reindented to aligned format. Default: FALSE
use_space_around_operators	Boolean; if TRUE spaces are placed around mathematical operators. Default: FALSE
wrap_after	Positive integer specifying the column limit (in characters) for wrapping comma-separated lists. If NULL, every item is put on its own line. Default: NULL
comma_first	Boolean; if TRUE a linebreak is inserted before comma. Default: FALSE

`truncate_strings`
: Positive integer; string literals longer than the given value are truncated. Default: `NULL`

`truncate_char`
: Character string appended if long string literals are truncated. Default: `"[...]"`

`encoding`
: Character string specifying the input encoding. Default: `NULL` (assumes UTF-8 or latin-1)

Details

This function is a wrapper to the *sqlparse.format()* function from the *sqlparse* Python module, which is a non-validating SQL parser.

Value

Character string containing the formatted SQL statements.

See Also

`sql_split`, `sql_parse`

Examples

```
if (reticulate::py_module_available("sqlparse")) {

  library("sqlparseR")

  raw <- "SELECT * FROM FOO WHERE BAR > 4500;"

  formatted <- sql_format(raw,
                          keyword_case = "capitalize",
                          identifier_case = "lower",
                          reindent = TRUE,
                          indent_after_first = TRUE)
  cat(formatted)
}
```

---

sql_parse *Parse SQL Statements*

Description

Parse one or several SQL statements (non-validating).

Usage

```
sql_parse(sql, encoding = NULL)
```

Arguments

sql	Character string containing one or more SQL statements to be formatted.
encoding	Character string specifying the input encoding. Default: NULL (assumes UTF-8 or latin-1)

Details

This function is a wrapper to the *sqlparse.parse()* function from the *sqlparse* Python module, which is a non-validating SQL parser.

Value

List with *reference class* objects which are converted instances of the custom Python class *Statement*. These tree-ish representations of the parsed statements can be analyzed with a set of reference class methods (accessed via $). See the documentation of the corresponding Python methods: https://sqlparse.readthedocs.io/en/stable/analyzing/.

See Also

sql_format, sql_split

Examples

```
if (reticulate::py_module_available("sqlparse")) {

  library("sqlparseR")

  raw <- "select*from foo; select*from bar;"

  parsed <- sql_parse(raw)

  ## Analyzing the parsed statements
  # e.g., get name of identifier in second statement
  n <- parsed[[2]]$get_name()
  print(n)

  # e.g., get a (Python) generator yielding ungrouped tokens of the first statement
  token_it <- parsed[[1]]$flatten()
  for (t in reticulate::iterate(token_it)) {
    print(t)
  }
}
```

`sql_split` *Split SQL to Single Statements*

Description

Split a string with (one or) several SQL statements into single statements.

Usage

```
sql_split(sql, encoding = NULL)
```

Arguments

`sql`	Character string containing (one or) several SQL statements
`encoding`	Character string specifying the input encoding. Default: `NULL` (assumes UTF-8 or latin-1)

Details

This function is a wrapper to the *sqlparse.split()* function from the *sqlparse* python module, which is a non-validating SQL parser.

Value

Character vector with the single SQL statements.

See Also

`sql_format`, `sql_parse`

Examples

```
if (reticulate::py_module_available("sqlparse")) {

  library("sqlparseR")

  raw <- "select*from foo; select*from bar;"

  statements <- sql_split(raw)

  print(statements)

}
```

Index