snowmobile

Release 0.2.0b25

Grant E Murray

Jan 25, 2022

CONTENTS

snowmobile bundles the SnowflakeConnection into an object model focused on configuration-management and streamlining access to Snowflake within Python.

Its main features are:

Note: snowmobile is a wrapper **around** the snowflake.connector, not a replacement for it; the SnowflakeConnection is intentionally stored as a public attribute so that the snowflake.connector and snowmobile APIs can be leveraged congruently.

CHAPTER

ONE

SETUP

1.1 1. Install

```
pip install snowmobile
```

1.2 2. Save *snowmobile.toml*

Download `snowmobile-template.toml` and save it in anywhere on your file system as **`snowmobile.toml`**.

1.3 3. Store Credentials

The first few lines of *snowmobile.toml* are outlined below; **for minimum configuration,** populate lines 6-12 with a valid set of Snowflake credentials.

```
2  [connection]
3      default-creds = ''
4
5      [connection.credentials.creds1]
6          user = ''
7          password = ''
8          role = ''
9          account = ''
10         warehouse = ''
11         database = ''
12         schema = ''
```

FYI: see here if unfamiliar with *.toml* syntax

More Info

On line **3**, `default-creds` enables specifying the default *alias* of the connection arguments to authenticate with by default if not specified in the *creds* parameter of `snowmobile.connect()`.

If left empty and also not provided as a parameter, arguments under the alias `creds1` will be authenticated with as it's the first set of credentials stored at the level of **`connection.credentials.*`**

See *Connector: Parameter Resolution* for details on how determines what gets passed to snowflake.connector.connect()

1.4 4. Connect to Snowflake

Successful setup and connection can be verified with:

```
import snowmobile

sn = snowmobile.connect()
"""
Looking for snowmobile.toml in local file system..
(1 of 1) Located 'snowmobile.toml' at ../Snowmobile/snowmobile.toml
..connected: snowmobile.Snowmobile(creds='creds1')
"""
```

Related: Executing Raw SQL Issues? See Docs

CHAPTER

TWO

SNOWMOBILE

An instance of `Snowmobile`, , represents a distinct session along with the contents of the *snowmobile.toml* with which it was instantiated.

Its purpose is to provide an entry point that will:

1. Locate, parse, and instantiate *snowmobile.toml* as a `Configuration` object, *`sn.cfg`*
2. Establish connections to Snowflake
3. Store the SnowflakeConnection, *`sn.con`*, and execute commands against the database

2.1 Usage

- *Connecting*
- *Executing Raw SQL*
- *Aliasing Credentials*
- *Parameter Resolution*
- *Delaying Connection*
- *Specifying snowmobile.toml*
- *Using* ensure_alive

Setup **This section assumes the following about the contents of *snowmobile.toml*:**

1. *[connection.credentials.creds1]* and *[connection.credentials.creds2]* are:
    1. Populated with valid credentials
    2. The first and second credentials stored respectively
    3. Aliased as *creds1* and *creds2* respectively
2. *default-creds* has been left blank

2.1.1 Connecting to Snowflake

Establishing a connection can be done with:

```
import snowmobile

sn = snowmobile.connect()
```

Here's some basic information on the composition of `sn`:

```
print(sn)             #> snowmobile.Snowmobile(creds='creds1')
print(sn.cfg)         #> snowmobile.Configuration('snowmobile.toml')
print(type(sn.con))   #> <class 'snowflake.connector.connection.SnowflakeConnection'>
```

Given ⚙, `sn` is implicitly using the same connection arguments as:

```
sn2 = snowmobile.connect(creds="creds1")
```

Here's some context on how to think about these two instances of *Snowmobile*:

```
sn.cfg.connection.current == sn2.cfg.connection.current  #> True
sn.current("schema") == sn2.sql.current("schema")     #> True
sn.current("session") == sn2.sql.current("session")   #> False
```

connecting.py

2.1.2 Executing Raw SQL

The following three methods are available for statement execution directly off .

sn.query()

```
df = sn.query("select 1")   #  == pd.read_sql()
type(df)                    #> pandas.core.frame.DataFrame
```

+

query() implements `pandas.read_sql()` for querying results into a `pandas.DataFrame`.

```
df = sn.query("select 1")   #  == pd.read_sql()
type(df)                    #> pandas.core.frame.DataFrame

# -- pd.read_sql() --
import pandas as pd

df2 = pd.read_sql(sql="select 1", con=sn.con)

print(df2.equals(df))   #> True
```

sn.ex()

```
cur = sn.ex("select 1")     #  == SnowflakeConnection.cursor().execute()
type(cur)                   #> snowflake.connector.cursor.SnowflakeCursor
```

+

ex() implements `SnowflakeConnection.cursor().execute()` for executing commands within a SnowflakeCursor.

```
cur = sn.ex("select 1")     #  == SnowflakeConnection.cursor().execute()
type(cur)                   #> snowflake.connector.cursor.SnowflakeCursor

# -- SnowflakeConnection.cursor().execute() --
cur2 = sn.con.cursor().execute("select 1")

print(cur.fetchone() == cur2.fetchone())  #> True
```

sn.exd()

```
dcur = sn.exd("select 1")   #  == SnowflakeConnection.cursor(DictCursor).execute()
type(dcur)                  #> snowflake.connector.DictCursor
```

+

exd() implements `SnowflakeConnection.cursor(DictCursor).execute()` for executing commands within DictCursor.

```
dcur = sn.exd("select 1")   #  == SnowflakeConnection.cursor(DictCursor).execute()
type(dcur)                  #> snowflake.connector.DictCursor

# -- SnowflakeConnection.cursor(DictCursor).execute() --
from snowflake.connector import DictCursor

dcur2 = sn.con.cursor(cursor_class=DictCursor).execute("select 1")

print(dcur.fetchone() == dcur2.fetchone())  #> True
```

executing.py

SnowflakeCursor / DictCursor

Note

The accessors `sn.cursor` and `sn.dictcursor` are **properties** of `Snowmobile` that return a new instance each time they are accessed. Depending on the intended use of SnowflakeCursor or DictCursor, it could be better to store an instance for re-referencing as opposed to repeatedly instantiating new instances off `sn`.

+

The below demonstrates the difference between calling two methods on the `cursor` property compared to on the same instance of SnowflakeCursor.

```
import snowmobile

sn = snowmobile.connect()

cur1 = sn.cursor.execute("select 1")
cur2 = sn.cursor.execute("select 2")

cursor = sn.cursor
cur11 = cursor.execute("select 1")
cur22 = cursor.execute("select 2")

id(cur1) == id(cur2)     #> False
id(cur11) == id(cur22)   #> True
```

connector_cursor_note.py

Naming Convention

Tip

The following convention of variable/attribute name to associated object is used throughout snowmobile's documentation and source code, including in method signatures:

- **sn**: `snowmobile.Snowmobile`
- **cfg**: `snowmobile.Configuration`
- **con**: snowflake.connector.SnowflakeConnection
- **cursor**: snowflake.connector.cursor.SnowflakeCursor

+

For example, see the below attributes of *`Snowmobile`*:

```
import snowmobile

sn = snowmobile.connect()

type(sn)          #> snowmobile.core.connection.Snowmobile
```

(continues on next page)

(continued from previous page)

```
type(sn.cfg)      #> snowmobile.core.configuration.Configuration
str(sn.cfg)       #> snowmobile.Configuration('snowmobile.toml')

type(sn.con)      #> snowflake.connector.connection.SnowflakeConnection
type(sn.cursor)   #> snowflake.connector.cursor.SnowflakeCursor
```

inspect_connector.py

2.1.3 Aliasing Credentials

The *default snowmobile.toml* contains scaffolding for two sets of credentials, aliased `creds1` and `creds2` respectively.

By changing `default-creds = ''` to `default-creds = 'creds2'`, *Snowmobile* will use the credentials from `creds2` regardless of where it falls relative to all the other credentials stored.

The change can be verified with:

```
import snowmobile

sn = snowmobile.connect()

assert sn.cfg.connection.default_alias == 'creds2', (
    "Something's not right here; expected default_alias =='creds2'"
)
```

verify_default_alias_change.py

2.1.4 Parameter Resolution

-

will look in the following three places to compile the connection arguments that it passes to snowflake.connector.connect() when establishing a connection:

1. *[connection.default-arguments]*
2. *[connection.credentials.alias_name]*
3. Keyword arguments passed to `snowmobile.connect()`

If the same argument is defined in more than one entry point, the **last** value found will take precedent; the purpose of this resolution order is to enable:

- Embedding connection arguments (e.g. timezone or transaction mode) within an aliased credentials block whose **values** differ from defaults specified in *[connection.default-arguments]*
- Superseding any connection parameters configured in *snowmobile.toml* with keyword arguments passed directly to `snowmobile.connect()`

Details

The way implements resolving connection parameters from multiple entry points is outlined below.

The *[connection.default-arguments]* and *[connection.credentials.alias_name]* are merged as the *connect_kwargs* property of *Connection* with:

```
    @property
    def connect_kwargs(self) -> Dict:
        """Arguments from snowmobile.toml for `snowflake.connector.connect()`."""
        return {**self.defaults, **self.current.credentials}
```

connect_kwargs is then combined with keyword arguments passed to `snowmobile.connect()` within the method itself as the `con` attribute of is being set:

```
    def connect(self, **kwargs) -> Snowmobile:
        """Establishes connection to Snowflake.
        ...
        """
        try:
            self.con = connect(
                **{
                    **self.cfg.connection.connect_kwargs,  # snowmobile.toml
                    **kwargs,  # any kwarg over-rides
                }
            )
            self.sql = sql.SQL(sn=self)
            print(f"..connected: {str(self)}")
            return self

        except DatabaseError as e:
            raise e
```

2.1.5 Delaying Connection

-

Sometimes it's helpful to create a `Snowmobile` without establishing a connection; this is accomplished with:

```
import snowmobile

sn = snowmobile.connect(delay=True)
```

When provided with `delay=True`, the that's returned omits connecting to Snowflake upon its instantiation; its `con` attribute is *None*, but its `cfg` attribute is a fully valid *Configuration* object.

See the tabbed *Examples* for more info.

Example: Implicit Connection

When provided with `delay=True`, the `con` attribute of will be *None* until a method is called on it that requires a connection.

If such a method is invoked, a call is made by to snowflake.connector.connect(), a connection established, and the attribute set.

```
import snowmobile

sn = snowmobile.connect(delay=True)

type(sn.con)      #> None
print(sn.alive)   #> False

_ = sn.query("select 1")

type(sn.con)      #> snowflake.connector.connection.SnowflakeConnection
print(sn.alive)   #> True
```

connector_delayed1.py

Example: Explicit Connection

In addition to implictly connecting by executing a query, the `connect()` method can be called on an existing instance of ; this will establish an initial connection if

was created with `delay=True` or a new session with the existing connection arguments otherwise.

```
import snowmobile

# -- Delayed Connection --
sn_del = snowmobile.connect(delay=True)

print(type(sn_del.con))   #> None
sn_del.connect()
print(type(sn_del.con))   #> snowflake.connector.connection.SnowflakeConnection


# -- Live Connection --
sn_live = snowmobile.connect()

session1 = sn_live.sql.current('session')
sn_live.connect()
session2 = sn_live.sql.current('session')
print(session1 != session2)   #> True
```

connector_delayed2.py

2.1.6 Specifying snowmobile.toml

From File Path

A full path (`pathlib.Path` or `str`) to a *snowmobile.toml* file can be provided to the `from_config` parameter to instantiate from a specific configuration file.

In practice, this looks like:

```
from pathlib import Path

import snowmobile

path = Path.cwd() / 'snowmobile_v2.toml'  # any alternate file path

sn = snowmobile.connect(from_config=path)
```

specifying_configuration.py

This will bypass any checks for a cached path and is useful for:

1. Testing different sets of configuration options without altering the original *snowmobile.toml* file
2. Binding a specific configuration with a process for sql-parsing purposes
3. Hard coding the configuration source in processes that have access to limited file systems (e.g. containers or VMs)

From File Name

Snowmobile caches locations based on the file **name** provided to the `config_file_nm` parameter of `snowmobile.connect()`, the default value of which is `snowmobile.toml`.

If an alternate file name is provided, it will be located and its location cached in the same way as the global *snowmobile.toml* file so that future instances of

on the same machine can make use of it upon instantiation without having to re-locate it.

The below codes are a contrived example demonstrating this behavior in practice.

Setup

All code blocks in this example are from *the same code file*, assumed to be executed in full starting with code directly below in which a second configuration file called *snowmobile2.toml* is created in the same folder as the global snowmobile.toml file.

```
import time
import shutil
import snowmobile

# Instantiate sn from snowmobile.toml; omit unnecessary connection
sn = snowmobile.connect(delay=True)

# Create alternate snowmobile.toml file called 'snowmobile2.toml'
path_cfg_orig = sn.cfg.location
path_cfg2 = path_cfg_orig.parent / 'snowmobile2.toml'
shutil.copy(path_cfg_orig, path_cfg2)
```

Below, `alt_sn()` is used to create `sn_alt1` and `sn_alt2`, representing an initial and future instance of respectively:

```
def alt_sn(n: int) -> snowmobile.Snowmobile:
    """Instantiate sn from snowmobile2.toml and print time elapsed."""
    pre = time.time()
    sn = snowmobile.connect(
        config_file_nm='snowmobile2.toml',
        delay=True  # omit connection - not needed
    )
    print(f"n={n}, time-required: ~{int(time.time() - pre)} seconds")
    return sn


sn_alt1 = alt_sn(n=1)  #> n=1, time-required: ~6 seconds  -> locates file, caches path
sn_alt2 = alt_sn(n=2)  #> n=2, time-required: ~0 seconds  -> uses cache from sn_alt1
"""
Note:
    The time required for `sn_alt1` to locate 'snowmobile2.toml' is arbitrary and
    will vary based the file's location relative to the current working directory.
"""
```

Cleanup is done with the following two lines which remove the *snowmobile2.toml* file created during the ⚙ for this example:

```
import os
os.remove(sn_alt1.cfg.location)
```

specifying_configuration2.py

2.1.7 Using ensure_alive

Controlling the behavior of `Snowmobile` when a connection is lost or intentionally killed is done through the `ensure_alive` parameter.

Its default value is *True,* meaning that if the `alive` property evaluates to *False*, **and a method is invoked that requires a connection,** it will re-connect to Snowflake before continuing execution.

Note

A re-established connection will not be on the same session as the original connection.

See *this snippet* for additional details.

CHAPTER

THREE

SCRIPT

`snowmobile.Script` parses a raw sql file into a composition of objects that can be leveraged for:

Documentation and standardization of sql

Access to individual statements within a script

Lightweight control flow and QA

Code generation and warehouse cleanup

3.1 Overview

Note: If you're just wanting to run some sql

The most straight-forward way to execute a local sql file is through the SnowflakeConnection.execute_stream() method, the API for which can be accessed from an instance of with:

```
import snowmobile

from codecs import open
```

(continues on next page)

(continued from previous page)

```
sn = snowmobile.connect()
with open(sqlfile, 'r', encoding='utf-8') as f:
   for cur in sn.con.execute_stream(f):
       for ret in cur:
           print(ret)
```

3.1.1 Model Intro

intro1.sql

This section creates a *Script* from the following file, *intro1.sql*, containing 3 bare sql statements:

```
-- ./docs/snippets/script/intro/intro1.sql

create or replace table sample_table (
  col1 number(18,0),
  col2 number(18,0)
);

insert into sample_table (col1, col2) values(1, 2);

select * from sample_table;
```

intro1.sql

Crash Course

Creating a Script

snowmobile.Script(path=path)

snowmobile.Script identifies sql and metadata in a sql file; assuming *path* is a full path to

intro1.sql , **script** can be created with:

```
import snowmobile

script = snowmobile.Script(path=path)
```

Each command is instantiated as its own *Statement* and stored according to its position in the original script; `script.dtl()` is used to send a summary of the contents parsed by **script** to the console:

```
script.dtl()
```

```
intro1.sql
==========
1: Statement('create table~s1')
2: Statement('insert into~s2')
3: Statement('select data~s3')
```

FYI

`script.dtl()` is generating its output with something like:

```
for i, s in script.items():
    print(f"{i}: {s}")
```

```
1: Statement('create table~s1')
2: Statement('insert into~s2')
3: Statement('select data~s3')
```

Instantiating from raw sql

Missing Content

Because these are bare sql statements..

```
s3 = script(3)
print(s3.index)      #> 3
print(s3.sql())      #> select * from sample_table
print(s3.kw())       #> select
print(s3.anchor())   #> select data
print(s3.desc())     #> s3
print(s3.nm())       #> select data~s3
```

Core Objects

When *Script* parses a string of sql, it identifies and stores *statements*, *tags*, and *markers*:

`Statement`*A valid sql command, a standard set of attributes, and any information (optionally) provided in a* tag

Tag*An arbitrary amount of information wrapped in a pre-defined, sql-compliant pattern that is parsable by snowmobile*

Marker*A collection of information within a* tag *that is associated with the script (or a subset of it) as opposed to an individual statement*

Note

The simple zen is to enable the consistent, clear annotation of sql in a way that is: **(1)** easily human-readable / writable **(2)** syntactically (& idiomatically) compliant **(3)** identifiable and parsable by snowmobile

To that end, *snowmobile.Script* intentionally ignores all comments that are not part of a *tag*.

Sections & Markup

A `Section` can be instantiated from a *`Statement`* or a *Marker*, and the *`Markup`* class combines multiple sections into a single document:

`Section`*Performs additional operations on the attributes from a* `Statement` *or a* Marker*, typically to generate a 'headered' section in a markdown file or a sql statement stripped of surrounding comments*

Markup*A context-specific collection of all sections within a script; capable of exporting markdown and tidied sql files*

Calling the *`doc()`* method on a *Script* will return a *`Markup`* of its contents.

The *`Markup.save()`* method will (by default) export a pair of files into a `.snowmobile` folder directly adjacent to the file with which the *Script* was instantiated.

A base case for this in practice is outlined in

Example: **intro.sql**

Note The following options can be configured on an instance of *`Markup`* prior to calling *`save()`*:

- Target location
- File names
- File types
- File contents

Example: intro.sql

```
/*-
__intro.sql__
__authored-by: Some Chap or Lass
__authored-on: Some Day or Year
__p*_***:
**Impetus**: *SQL is older than time and isn't going anywhere; might we allow a␣
↪simple markup syntax?*
-*/

/*-
create table~sample_table; DDL
__description: This is an example statement description
-*/
create or replace table sample_table (
        col1 number(18,0),
        col2 number(18,0)
);
```

intro.sql

With a `path` to *intro.sql*, the following can be run:

```
import snowmobile

script = snowmobile.Script(path=path)
markup = script.doc()

print(script)   #> snowmobile.Script('intro.sql')
print(markup)   #> snowmobile.core.Markup('intro.sql')

markup.save()
```

Given *intro.sql* is here:

```
sql/
└─ intro.sql
```

markup.save() created the `.snowmobile` directory and exported the following files:

```
sql/
├─ intro.sql
└─ .snowmobile/
   ├─ intro.md
   └─ intro.sql
```

intro.md

- **Authored-By**: *Some Chap or Lass*
- **Authored-On**: *Some Day or Year*

Impetus: *SQL is older than time and isn't going anywhere; might we allow a simple markup syntax?*

- **Description**: *This is an example statement description*

```
create or replace table sample_table (
    col1 number(18,0),
    col2 number(18,0)
);
```

intro.sql

```
/*: -------------------------------------------------------------------------
    ** This file was stripped of all comments and exported by Snowmobile **
    ------------------------------------------------------------------------- :*/

/*-
__intro.sql__
__authored-by: Some Chap or Lass
__authored-on: Some Day or Year
__p*_***:
**Impetus**: *SQL is older than time and isn't going anywhere; might we allow a␣
↪simple markup syntax?*
-*/

/*-create table~sample_table; DDL-*/
create or replace table sample_table (
```

(continues on next page)

(continued from previous page)

```
        col1 number(18,0),
        col2 number(18,0)
);
```

3.1.2 Statements

script

This section performs operations on the following :

```
script = snowmobile.Script(path=path)
```

Where `path` (`pathlib.Path` or `str`) is a full path to *overview.sql*.

The 7 generic sql statements within *overview.sql* are arbitrary and chosen based only on the loose criteria of:

1. Includes the minimum variety of Statements and *Markup* to demonstrate the fundamentals of how *Script* parses sql
2. Is executable from top to bottom without requiring external setup

overview.sql

```
create or replace table sample_table (
        col1 number(18,0),
        col2 number(18,0)
);

insert into sample_table with
sample_data as (
  select
    uniform(1, 10, random(1)) as rand_int
  from table(generator(rowcount => 3)) v
)
  select
    row_number() over (order by a.rand_int) as col1
    ,(col1 * col1) as col2
  from sample_data a;

select * from sample_table;

/*-select all~sample_table-*/
select * from sample_table;

create or replace transient table any_other_table clone sample_table;

insert into any_other_table (
  select
    a.*
  from sample_table a
);

drop table if exists sample_table;
```

Intro

When a sql file is parsed by `Script`, each statement is identified and instantiated as its own *`Statement`*.

An overview of the statements within a script's context can be sent to the console with `script.dtl()`; in the case of , this looks like:

```
script.dtl()
```

```
overview.sql
============
1: Statement('create table~s1')
2: Statement('insert into~s2')
3: Statement('select data~s3')
4: Statement('select all~sample_table')
5: Statement('create transient table~s5')
6: Statement('insert into~s6')
7: Statement('drop table~s7')
```

Accessing the first and last statements of and inspecting a few of their attributes can be done with:

```
# Store a few st, accessed by index position
s_first, s_last = script(1), script(-1)

# first sql keyword
print(s_first.kw)  #> create
print(s_last.kw)   #> drop

# position within `script`
print(s_first.index)  #> 1
print(s_last.index)   #> 7
```

A *`Statement`* can be interacted with off the *Script* or stored and used independently; for example, here are two ways that the first statement in *overview.sql* can be executed:

```
script.run(1)     # .run() from `script`
script(1).run()   # .run() from `statement`
```

Those above are several amongst a set of *`Statement`* attributes that can be used to alter the scope of a *Script*.

For example, the following snippet filters out `drop` and `select` statements based on their `kw` attribute and returns a modified , `s`, that can be operated on within that context:

```
print(script.depth)   #> 7
print(script(1).nm)   #> create table~s1
print(script(-1).nm)  #> drop table~s7

with script.filter(excl_kw=['select', 'drop']) as s:
    print(s.depth)   #> 4
    print(s(1).nm)   #> create table~s1
    print(s(-1).nm)  #> insert into~s4
    s.dtl()
```

```
overview.sql
============
```

(continues on next page)

(continued from previous page)

```
1: Statement('create table~s1')
2: Statement('insert into~s2')
3: Statement('create transient table~s3')
4: Statement('insert into~s4')
```

overview-statement-intro.py The following section outlines how these components are constructed.

Statement Names

The intent of the following taxonomy is to define a standard such that the name for a given statement is:

1. Constructed from attributes that can be unambiguously parsed from a piece of raw sql
2. Structured such that user *provided* names can be easily implemented and loosely parsed into the same set of attributes as those *generated* from (**1**)

Every statement has a `Name` with a set of underlying properties that are used by the rest of the API; for each property, there is a *generated* (**_ge**) and *provided* (**_pr**) attribute from which its final value is sourced.

Generated attributes are populated for all st, whereas only those with a name specified in a *tag* have populated *provided* attributes; consequently, a *provided* value takes precedent over its *generated* counterpart.

Example: nm

The `nm` value for a given statement will be equivalent to its `nm_pr` if present and its `nm_ge` otherwise.

This resolution order is repeated across the underlying components of `nm`, documented in the following sections.

---

s1 & s4

The below statements, `s1` and `s4`, from are used throughout the remaining examples in *this section*.

```
# Store statements 1 and 4 for inspection
s1, s4 = script(1), script(4)
```

---

nm

{anchor}{delimiter}{desc}

`anchor`*what operation is a statement performing*

`delimiter`

a configured value

with which to delimit the `anchor` *and* `desc`

`desc`*A free-form piece of text associated with the statement*

-

`nm` is the highest-level accessor for a *`Statement`*.

Its values for s1 & s4 (for example) can be inspected with:

```
print(s1.nm)       #> create table~s1
print(s4.nm)       #> select all~sample_table
```

nm_pr

In determining the `nm` for s1 specifically, is considering the following two lines of *overview.sql*:

```
/*-select all~sample_table-*/
select * from sample_table;
```

Each of these two lines above is the respective source for *provided* and *generated* information about the statement called out in Example: **nm** , the underlying values for which can be inspected in the same way:

```
print(s4.anchor_ge)  #> select data
print(s4.anchor_pr)  #> select all
print(s4.anchor)     #> select all

print(s4.desc_ge)    #> s4
print(s4.desc_pr)    #> sample_table
print(s4.desc)       #> sample_table

print(s4.nm_ge)      #> select data~s4
print(s4.nm_pr)      #> select all~sample_table
print(s4.nm)         #> select all~sample_table
```

anchor

{kw} {obj}

`kw`*the literal first sql keyword the statement contains*

`obj`*the in-warehouse object found in the first line of the statement*

-

`anchor` represents all text to the left of the first *`delimiter`* and when generated will fit the above structure to a varying degree depending on the sql being parsed and configurations in *snowmobile.toml*.

For s1 & s4 :

```
print(s1.anchor)  #> create table
print(s4.anchor)  #> select all
```

kw

-

`kw` is the literal first *keyword* within the command being executed by a statement's sql.

For s1 & s4:

```
print(s1.kw)   #> create
print(s4.kw)   #> select
```

keyword-exceptions

The *keyword-exceptions* section in the *[sql]* block of *snowmobile-ext.toml* enables specifying an alternate keyword for a literal keyword parsed from a statement's sql; alternate keywords will populate the statement's`skw_ge` as opposed to the literal keyword identified at the start of the statement:

```
   [sql.keyword-exceptions]
       "with" = "select"
```

The default included above is the reason that the `kw` for both the following statements is `select` as opposed to `select` and `with` respectively:

```
-- kw = 'select'
select * from any_table;

-- kw = 'select'
with some_cte as (
 select * from any_table
)
 select * from some_cte;
```

obj

-

`obj` is determined by a case-insensitive, full ('word-boundaried') search through the **first** line of a statement's sql for a match within a pre-defined set of values.

named-objects

The values for which a match is checked are configured in the *named-objects* section within the *[sql]* block of *snowmobile-ext.toml*, included below.

Matching is peformed against values in the **literal** order as they are configured in *snowmobile-ext.toml* until a match is found or the list is exhausted; it is enforced that the object found cannot be equal to the `kw` for the statement.

```
   named-objects = [
       # 'grant' statements          "select",
       "all",
       "drop",

       # base objects
```

(continues on next page)

(continued from previous page)

```
        "temp table",
        "transient table",
        "table",
        "view",
        "schema",
        "warehouse",
        "file format",

        # plural bases
        "tables",
        "views",
        "schemas",
    ]
```

Note

The above order is as such so that table qualifiers for the following three (types of) statements are reflected in the `obj` for each.

```
-- obj = 'table'
create table any_table as
select 1 as any_col;

-- obj = 'transient table'
create transient table any_table2 as
select 1 as any_col;

-- obj = 'temp table'
create temp table any_table3 as
select 1 as any_col;
```

generic-anchors

A mapping of sql keywords to generic anchor names can be configured in the *generic-anchors* block within the *[sql]* section of *snowmobile-ext.toml*, included below.

```
[sql.generic-anchors]
    "select" = "select data"
    "set" = "set param"
    "unset" = "unset param"
    "insert" = "insert into"
    "delete" = "delete from"
```

delimiter

-

delimiter is a literal constant specified in the description-delimiter field within the *[script.patterns.core]* section of *snowmobile.toml*, the value for which can be accessed directly off with:

```
print(sn.cfg.script.patterns.core.delimiter)  #> ~
```

desc

-

desc is a free-form text field loosely intended to be short-hand *description* for the statement.

The **generated** description for a statement, desc_ge, is a concatenation of a constant prefix and its index position within the script.

The prefix used is configurable in the description-index-prefix field within the *[script.patterns.core]* section of *snowmobile.toml*, the value for which can be accessed directly off with:

```
print(sn.cfg.script.patterns.core.prefix)  #> s
```

The **provided** description for a statement, desc_pr, is all text to the right of the first *character* found matching the *delimiter* within a statement's nm_pr.

using *desc-is-simple*

Warning

The functionality outlined below is experimental and not under test.

Using parsed values for the obj_ge and desc_ge can be enabled by setting the *desc-is-simple* field to true in *snowmobile-ext.toml* or by modifying the attribute's value on an instance of .

In the case of , this looks like:

```
# alter default value of 'desc_is_simple'
sn.cfg.sql.desc_is_simple = False

# re-inspect the script's contents
script.dtl()
```

```
overview.sql
============
1: Statement('create table~sample_table: s1')
2: Statement('insert into~sample_table: s2')
3: Statement('select data~sample_table: s3')
4: Statement('select all~sample_table')
5: Statement('create transient table~any_other_table clone sample_table: s5')
6: Statement('insert into~any_other_table: s6')
7: Statement('drop table~sample_table: s7')
```

3.1.3 Tags

A tag contains an abitrary amount of information wrapped in a pre-defined opening/closing pattern. It can be associated with a *Statement*, identified by its literal position relative to the statement's sql, or with a *Marker*, identified by its contents.

The default pattern, highlighted in the below snippet from *snowmobile.toml*, mirrors that of a standard sql block comment with an additional dash (-) on the inside of each component:

```
64    export-dir-name = '.snowmobile'
65    result-limit = -1
```

3.1.4 Markers

Overview

TODO

Missing

+-

MORE CONTENT GOES HERE

3.1.5 Markup

Using markup within a script enables:

- Defining accessors for individual statements - Adding descriptive information to individual statements or to the script itself
- Finer-grained control of the script's execution
- Generating documentation and cleansed sql files from the working version of a script

snowmobile introduces two sql-compliant forms of adding markup to a sql file:

1. *Tags* enable constructing collections of attributes amidst sql st, including those directly associated with a particular statement
2. *Markers* are a collection of attributes that are **not** associated with a particular statement

The following sections outline the different ways that *Tags* and *Markers* are implemented and utilized.

Single-Line Tags

Overview

Single-line tags are the simplest form of *markup* and can be used to succinctly denote a name for a given statement.

When a single-line string directly precedes a statement and is wrapped in *a valid open/close pattern*, it will be recognized as the *provided* name (`nm_pr`) and used as the statement's name (`nm`) as opposed to its *generated* name (`nm_ge`).

+

Consider the sql file, *tags_single-line.sql*, containing two st, the first and second of which have valid and invalid single-line tags respectively:

```
-- ..docs/snippets/script/tags_single-line.sql

/*-I am a wrap-*/
select * from sample_table;

/*-I am a wrap that isn't positioned correctly-*/

select * from sample_table;
```

Given a `path` to *tags_single-line.sql* and , the following `script` can be created:

```
# Instantiate a Script from sql file
script = snowmobile.Script(path=path, sn=sn)

# Store individual statements for inspection
s1, s2 = script(1), script(2)

print(s1)         #> Statement('I am a tag')
print(s1.nm_ge)   #> select data~s1
print(s1.nm_pr)   #> I am a tag
print(s1.nm)      #> I am a tag

print(s2)         #> Statement('select data~s2')
print(s2.nm_ge)   #> select data~s2
print(s2.nm_pr)   #> ''
print(s2.nm)      #> select data~s2
```

Note

The first statement has a valid tag directly preceding it, so its name (`nm`) is populated by the *provided* name within the tag (`nm_pr`) as opposed to the name that was *generated* for the statement (`nm_ge`).

The second statement does **not** have a valid tag directly preceding it, so its generated name, `select data~s2`, is used and the line `/*-I am a tag that isn't positioned correctly-*/` is ignored.

Multi-Line Tags

Overview

Multi-line tags provide a method of associating multiple attributes with a *Statement* according to the following syntax:

- Attribute **names** must:
  1. Start at the beginning of a new line
  2. Have leading double underscores (__)
  3. End with a single colon (:)
- Attribute **values** have no restrictions except for several reserved attributes documented in the *reserved attributes* (LINK NEEDED) section below

+

In practice, this looks something like the following:

```
-- ..docs/snippets/script/tags_multi-line.sql

/*-
__name: I am a wrap
__description: This is an example of a wrap with the name explicitly declared.
-*/
select * from sample_table;

/*-
I am another wrap
__description: This is an example of a wrap with the name implicitly declared.
-*/
select * from sample_table;
```

Tip

Trailing wildcards can be appended to attribute **names** to denote how information will be rendered in generated documentation; this is covered in *Patterns - Wildcards* below.

Patterns

TODO

Missing

+-

MORE CONTENT GOES HERE

Core

Overview

TODO

Missing

+-

MORE CONTENT GOES HERE

Wildcards

Overview

TODO

Missing

+-

MORE CONTENT GOES HERE

CHAPTER

FOUR

TABLE

Note

See *the snowmobile.core.table API docs*

test

CHAPTER

FIVE

SQL

inherits all methods of a *SQL* class that generates and executes raw SQL from inputs; its purpose is to provide a simple, on-hand Python API for querying metadata and executing basic administrative commands against Snowflake.

By default, `sql` will execute the generated sql and return its results; execution can be omitted and the generated sql returned as a raw string by providing *run=False* to the method being invoked or by manually setting its `auto_run` attribute to *False* prior to calling the method.

Warning These methods will not ask twice before querying, altering or dropping a Snowflake object; isolated testing to ensure the API is understood before use is recommended.

Providing *run=False* and printing the returned string to the console is one of the easiest ways to inspect the sql that's generated by a given method.

```
print(sn.drop('sample_table', run=False))
"""
>>>
drop table if exists sample_table
"""
```

5.1 Usage

- *Command Overview*
- *Execution Control*
- *Setting nm and obj*

Setup

These examples make useof a **sample_table** containing:

COL1	COL2
1	1
2	4
4	9

5.1.1 Command Overview

-

FYI

The snippets below encompass the most widely applicable methods available off *snowmobile.SQL*; see the API Docs for exhaustive method documentation.

The following statements can be run to interact with **sample_table** *defined by* ⚙.

Verify it exists:

```
sn.exists('sample_table')  #> True
```

Sample its records:

```
sn.select('sample_table', n=1)
```

Query its columns from selecting a sample record:

```
sn.columns('sample_table')  #> ['COL1', 'COL2']
```

Or from the information schema:

```
sn.columns('sample_table', from_info_schema=True)  #> ['COL1', 'COL2']
```

Check its depth:

```
sn.count('sample_table')  #> 3
```

Query its DDL:

```
print(sn.ddl('sample_table'))
"""
>>>
create or replace TABLE SAMPLE_TABLE (
  COL1 FLOAT,
  COL2 FLOAT
);
"""
```

Clone it to another table:

```
sn.clone(nm='sample_table', to='sample_table2')
```

Drop objects:

```
sn.drop('sample_table2')
sn.exists('sample_table2')  #> False
```

Cross-Schema

Applicable methods of `sql` inspect the value passed through the `nm` argument for schema-prefixes; when provided, `sn` will compare the schema passed as an argument to the schema associated with in order to generate the appropriate sql.

For example, if `other_schema` represents a different schema than is currently connected to, the following two statements could be run:

Clone `sample_table` to `other_schema.sample_table`:

```
sn.clone(nm='sample_table', to='other_schema.sample_table')
```

Drop `other_schema.sample_table` from the current schema:

```
sn.drop(nm='other_schema.sample_table')
```

5.1.2 Execution Control

Also demonstrated *above*, methods can be provided with *run=False* to return the raw sql as a string as opposed to executing the generated command:

```
print(sn.drop('sample_table', run=False))
"""
>>>
drop table if exists sample_table
"""

print(sn.select('sample_table', n=1))
"""
>>>
select
    *
from sample_table
limit 1
"""
```

The `run` method argument has the following signature:

run: Optional[`bool`] = None

If a valid `bool` isn't passed in the place of *None*, the current value of its *`auto_run`* attribute determines whether or not to execute the sql it generates.

An alternative to providing *run=False* across a series of methods in order to inspect the sql being generated is then to modify this attribute's value on a given instance of , done with:

```
sn.auto_run = False
```

Once set to *False*, an equivalent `sample1` and `sample2` can be created with:

```
sample1 = sn.select('sample_table', run=False)
sample2 = sn.select('sample_table')
```

(continues on next page)

(continued from previous page)

```
print(type(sample1))        #> <class 'str'>
print(sample1 == sample2)   #> True
```

Because methods defer to *auto_run* in absence of an explicit argument, it can be executed off the same instance of with:

```
df_sample = sn.select('sample_table', run=True)
print(type(df_sample))  #> <class 'pandas.core.frame.DataFrame'>
```

5.1.3 Setting `nm` and `obj`

Most `SQL` methods need to know an in-warehouse object's name (`nm`) and type (`obj`), which default to *None* and *table* respectively.

These defaults are why we can write:

```
_ = sn.drop('sample_table', run=False)
```

Instead of:

```
_ = sn.drop('sample_table', obj='table', run=False)
```

In the same way as the `run` method argument and the ***auto_run*** attribute, `SQL` deffers to the values of its `nm` and `obj` attributes in absence of valid strings passed through the `nm` and `obj` method arguments.

Bringing these together and assuming a default instace of , the following can be run:

```
sn.auto_run = False
sn.nm = 'sample_table'

sample1 = sn.select('sample_table')
sample2 = sn.select()
df_sample = sn.select(run=True)

print(type(sample1))        #> <class 'str'>
print(sample1 == sample2)   #> True
print(type(df_sample))  #> <class 'pandas.core.frame.DataFrame'>
```

CHAPTER

SIX

SNOWMOBILE.TOML

The parsed and validated form of *snowmobile.toml* is a `Configuration` object.

All parsing of the file is done within `snowmobile.core.cfg`, in which sections are split at the root and fed into pydantic's glorious API to define the schema and impose (evolving) validation where needed.

Once validated, the `Configuration` object serves as a namespace for the contents/structure of the configuration file and utility methods implemented on top of them, with the rest of the API accessing it as the `cfg` attribute of .

6.1 Inspecting `sn.cfg`

The `Configuration` model is accessed as the `cfg` attribute of `Snowmobile`; a straight-forward way to inspect its composition is to instantiate a *delayed instance* of :

```
import snowmobile

sn = snowmobile.Snowmobile(delay=True)

type(sn.cfg)               #> snowmobile.core.configuration.Configuration
print(sn.cfg.location)     #  'path/to/your/snowmobile.toml'
```

The following attributes of `sn.cfg` map to the root configuration sections of *snowmobile.toml*:

```
type(sn.cfg.connection)    #> snowmobile.core.cfg.connection.Connection
type(sn.cfg.loading)       #> snowmobile.core.cfg.loading.Loading
type(sn.cfg.script)        #> snowmobile.core.cfg.script.Script
type(sn.cfg.sql)           #> snowmobile.core.cfg.other.SQL
type(sn.cfg.ext_sources)   #> snowmobile.core.cfg.other.Location
```

inspect_configuration.py

Tip

The usage documentation contains detail on how changes to *snowmobile.toml values* flow through to

and impact the its implementation.

6.2 Glossary

`[connection]` *Configuration options used by when establishing connections to Snowflake*

`default-creds` *The credentials (alias) to use by default in absence of one provided to the* `creds` *keyword argument to* `snowmobile.connect()`

`[connection.credentials]` *Groups subsections of credentials, each declared with the structure of* `[connection.credentials.credentials_alias]`

`[connection.credentials.creds1]` *Store your first set of credentials here;* `creds1` *is a credentials alias*

`[connection.credentials.creds2]` *Store as many credentials as you want following this format; aliases must be unique*

`[connection.default-arguments]` *Credentials-agnostic keyword arguments to pass to snowflake.connector.connect()*

`[loading]` *Configuration options for data loading used by* snowmobile.Table

`[loading.default-table-kwargs]` *Default specifications for a* ***snowmobile.Table*** *object*

`[loading.put]` *Default arguments to include in Snowflake's put file from stage command*

`[loading.copy-into]` *Default arguments to include in Snowflake's copy into table command*

`[loading.save-options]` *Groups subsections of save-options*

`[loading.save-options."snowmobile_default_csv"]` *Default file-save options for* ***snowmobile_default_csv***

`[loading.save-options."snowmobile_default_psv"]` *Default file-save options for* ***snowmobile_default_psv***

`[external-sources]` *Defines paths to custom sources referenced by different snowmobile objects*

`ddl` *Posix path to a sql file containing DDL for file formats*

`extension` *Posix path to* snowmobile-ext.toml

`[script]` *Configurations for snowmobile.Script*

`export-dir-name` *Directory name for generated exports (markup and stripped sql scripts)*

`[script.patterns.core]` *Core patterns used for markup identification*

`open-tag` *Open-pattern for in-script tags*

`close-tag` *Close-pattern for in-script tags*

`description-delimiter` *Delimiter separating description from other statement attributes*

`description-index-prefix` *String with which to prepend a statement's index position when deriving* `desc_ge`

`[script.patterns.wildcards]` *Defines wildcards for attribute names within script tags*

`wildcard-character` *The literal character to use as a wildcard*

`wildcard-delimiter` *The literal character with which to delimit wildcards*

`denotes-paragraph` *Indicates the attribute* ***value*** *should be rendered as free-form markdown as opposed to a plain text bullet*

`denotes-no-reformat` *Indicates the attribute* ***name*** *should be left exactly as it is entered in the script as opposed to title-cased*

`denotes-omit-name-in-output` *Indicates to omit the attribute's **name** in rendered output*

`[script.qa]` *Default arguments for **QA-Diff** and **QA-Empty** Statements*

`partition-on` *Pattern to identify the field on which to partition data for comparison*

`compare-patterns` *Pattern to identify fields being compared*

`ignore-patterns` *Pattern to identify fields that should be ignored in comparison*

`end-index-at` *Pattern to identify the field marking the last index column*

`[script.qa.default-tolerance]` *Default values for QA-Delta tolerance levels*

`relative` *Default relative-difference tolerance*

`absolute` *Default absolute-difference tolerance*

`[script.markdown]` *Configuration for markdown generated from .sql files*

`default-marker-header` *Header level for markers (h1-h6)*

`default-statement-header` *Header level for statements (h1-h6)*

`default-bullet-character` *Character to use for bulleted lists*

`wrap-attribute-names-with` *Character to wrap attribute **names** with*

`wrap-attribute-values-with` *Character to wrap attribute **values** with*

`include-statement-index-in-header` *Denotes whether or not to include a statement's relative index number in its header along with its name*

`limit-query-results-to` *Maximum number of rows to include for a statement's rendered **Results***

`[script.markdown.attributes]` *Configuration options for specific attributes*

`[script.markdown.attributes.markers]` *Pre-defined marker configurations*

`[script.markdown.attributes.markers."__script__"]` *Scaffolding for a template marker called '__script__'*

`as-group` *The literal text within which to group associated attributes as sub-bullets*

`team` *A sample attribute called 'team'*

`author-name` *A sample attribute called 'author-name'*

`email` *A sample attribute called 'email'*

`[script.markdown.attributes.markers."__appendix__"]` *Scaffolding for a second template marker called '__appendix__'*

`[script.markdown.attributes.reserved.rendered-sql]` *Configuration options for a reserved attribute called 'rendered-sql'*

`include-by-default` *Include attribute by default for each* `Section`

`attribute-name` *The attribute's name as it is declared within a* tag

`default-to` *The attribute name as it should be interpreted when parsed*

`[script.markdown.attributes.reserved.query-results]` *Configuration for a reserved attributes called `query-results`*

`include-by-default` *Include attribute by default for each* `Section`

`attribute-name` *The attribute's name as it is declared within a* tag

`default-to` *The attribute name as it should be interpreted when parsed*

`format` *Render format for the tabular results; markdown or html*

`[script.markdown.attributes.from-namespace]` *List of* `Statement` *attributes to include in its* `Section`*; includes non-default attributes set on an instance*

`[script.markdown.attributes.groups]` *Defines attributes to be grouped together within a sub-bulleted list*

`[script.markdown.attributes.order]` *Order of attributes within a* `Statement`*-level section*

`[script.tag-to-type-xref]` *Maps tagged attributes to data types; will error if an attribute included here cannot be parsed into its specified data type*

`[sql]` *SQL parsing specifications for a* `Statement`

`provided-over-generated` *`nm_pr` takes precedent over `nm_ge`*

`desc-is-simple` *`True` invokes additional parsing into `desc` and `obj`*

`named-objects` *Literal strings to search for matches that qualify as a Snowflake object if included within the first line of a statement's sql and not equal to its first keyword*

`generic-anchors` *Generic anchors to use for a given keyword; will be used for generated statements if* `desc-is-simple` *is* **True**

`keyword-exceptions` *Alternate mapping for first keyword found in a command*

`information-schema-exceptions` *Map Snowflake objects to their `information_schema.*` table name if different than the plural form of the object; (e.g. `schema` information is in `information_schema.schemata` not `information_schema.schemas`)*

6.3 File Contents

```
[connection]
    default-creds = ''

    [connection.credentials.creds1]
        user = ''
        password = ''
        role = ''
        account = ''
        warehouse = ''
        database = ''
        schema = ''

    [connection.credentials.creds2]
        user = ''
        password = ''
        role = ''
        account = ''
        warehouse = ''
        database = ''
        schema = ''

    [connection.default-arguments]
        autocommit = true
        authenticator = 'snowflake'

[loading]
```

(continues on next page)

(continued from previous page)

```
    [loading.default-table-kwargs]
        file_format = 'snowmobile_default_psv'
        validate_table = true
        validate_format = true
        if_exists = 'append'
        keep_local = false
        reformat_cols = true
        upper_case_cols = true
        check_dupes = true
        load_copy = true

    [loading.put]
        auto_compress = true

    [loading.copy-into]
        on_error = 'continue'

    [loading.save-options]
        [loading.save-options."snowmobile_default_csv"]
            index = false
            header = false
            quotechar = '"'
            sep = ","
        [loading.save-options."snowmobile_default_psv"]
            index = false
            header = false
            quotechar = '"'
            sep = "|"

[external-sources]
    ddl = ''
    extension = ''
    sql-save-heading = ''

[script]
    export-dir-name = '.snowmobile'
    result-limit = -1

    [script.patterns]

        [script.patterns.core]
            open-tag = '/*-'
            close-tag = '-*/'
            description-delimiter = '~'
            description-index-prefix = "s"

        [script.patterns.markup]
            wildcard-character = '*'
            wildcard-delimiter = '_'
            denotes-paragraph = '*'
            denotes-no-reformat = '**'
            denotes-omit-name-in-output = '***'

    [script.qa]
        partition-on = 'src_description'
        compare-patterns = ['.*_diff']
```

(continues on next page)

(continued from previous page)

```
        ignore-patterns = ['.*_tmstmp']
        end-index-at = 'end_index'

        [script.qa.default-tolerance]
            relative = 0.0
            absolute = 0.0

    [script.markup]
        default-marker-header = 'h1'
        default-statement-header = 'h2'
        default-bullet-character = '*'
        wrap-attribute-names-with = '**'
        wrap-attribute-values-with = '_'
        include-statement-index-in-header = true
        limit-query-results-to = 20
```

6.4 snowmobile-ext.toml

```
# ================================================================================
# ../snowmobile-ext.toml
# DO NOT DELETE UNLESS ALTERNATE EXTENSION FILE IS SPECIFED IN SNOWMOBILE.TOML
# ================================================================================

# todo: + 'tabs-to-spaces' and 'tab-size' for reserved attributes.sql

# -- Configuration options for snowmobile.script() ----------------------------
[script]

    [script.markup.attributes]

        [script.markup.attributes.markers]
            [script.markup.attributes.markers."__script__"]
                as-group = 'Author Information'
                # ====/ start-attributes /====
                team = 'Sample Team Name'
                email = 'first.last@domain.com'

            [script.markup.attributes.markers."__appendix__"]
                as-group = ''
                # ====/ marker-attributes /====

        [script.markup.attributes.reserved.rendered-sql]
            # The literal sql for a given statement.
            include-by-default = true
            attribute-name = 'sql'
            default-to = 'SQL***'
            # TODO: Make each of these a dictionary of derived classes like QA for␣
↪Script
            #   with the .process() method being all the operations that have access␣
↪to
            #   the tagged values from the script and the arguments in this block
            # tabs-to-spaces = true
            # tab-size = 2

```

(continues on next page)

(continued from previous page)

```
        [script.markup.attributes.reserved.query-results]
            # The literal results returned by a given statement.
            include-by-default = false
            attribute-name = 'results'
            default-to = 'results*_***'
            format = 'markdown'
            tabulate-format = 'grid'

        [script.markup.attributes.from-namespace]
            # TODO: period separated values for nested vals to vars(obj)[k]
            #   :: execution.time.str to check if str in obj.callables or obj.wrap
            #   :: * only applying if the bool(val) that's returned is True
            execution_time_txt = 'Execution Time'
            outcome_txt = 'Last Outcome'

        [script.markup.attributes.groups]
            "Execution-Information**" = [
                # todo
                #   :: 'Execution Time', 'Last Outcome', etc
                'execution_time_txt',
                'outcome_txt'
            ]
            "QA-Specifications**" = [
                'partition-on',
                'end-index-at',
                'compare-patterns',
                'ignore-patterns',
                'absolute-tolerance',
                'relative-tolerance',
            ]

        [script.markup.attributes.order]
            attribute-order = [
                'authored-date',
                'author-information',
                'execution-information',
                'qa-specifications',
                'outcome_txt',
                'execution_time_txt',
                'description',
                'p',
                'sql',
                'results'
            ]


# -- Type mapping of how attribute values should be parsed based on attr name -
[script.tag-to-type-xref]
    string = [
        'name', 'partition-on', 'end-index-at', 'description'
    ]
    list = [
        'compare-patterns', 'ignore-patterns'
    ]
    float = [
        'absolute-tolerance', 'relative-tolerance'
    ]
```

(continues on next page)

(continued from previous page)

```
    bool = [
        'results', 'sql', 'transpose'
    ]

# -- Named object, generic anchors, and keyword-exceptions for sql parsing ----
[sql]
 provided-over-generated = true
    desc-is-simple = true

    named-objects = [
        # 'grant' statements
        "select",
        "all",
        "drop",

        # base objects
        "temp table",
        "transient table",
        "table",
        "view",
        "schema",
        "warehouse",
        "file format",

        # plural bases
        "tables",
        "views",
        "schemas",
    ]

    [sql.generic-anchors]
        "select" = "select data"
        "set" = "set param"
        "unset" = "unset param"
        "insert" = "insert into"
        "delete" = "delete from"

    [sql.keyword-exceptions]
        "with" = "select"

    [sql.information-schema-exceptions]
        schema = "schemata"
```

CHAPTER

SEVEN

SNOWMOBILE.CORE

`snowmobile` lives in `snowmobile.core` to keep from cluttering intellisense/autocomplete while interacting with the API.

7.1 Subpackages

7.1.1 snowmobile.core.cfg

Full configuration object model; represents a parsed `snowmobile.toml` file.

Submodules

snowmobile.core.cfg.connection

[connection] section from **snowmobile.toml**, including subsections.

Module Contents

Classes

Credentials	[connection.credentials.credentials_alias]
Connection	[connection]

class snowmobile.core.cfg.connection.**Credentials**

Bases: snowmobile.core.cfg.base.Base

[connection.credentials.credentials_alias]

user :str

password :str

role :str

account :str

warehouse :str

database :str

schema_name :str

as_nm(*self*, *n: str*)
Sets the credentials alias.

property credentials(*self*)
Returns namespace as a dictionary, excluding `_alias`.

class snowmobile.core.cfg.connection.**Connection**(***data*)
Bases: snowmobile.core.cfg.base.Base

[connection]

This includes the *default_alias* which is the set of credentials that `snowmobile` will authenticate with if *creds* is not explicitly passed.

default_alias
The set of credentials that is used if *creds* is not explicitly passed to `snowmobile.connect` on instantiation.

Type str

creds
The name given to the set of credentials within the **credentials** block of the **snowmobile.toml** file (e.g. [credentials.creds] assigns an *creds* to a given set of credentials.

Type str

creds
A dictionary of *creds* to the associated `Creds` object containing its credentials.

Type dict[str, Creds]

default_alias :str

provided_alias :str

credentials :Dict[str, Credentials]

defaults :Dict

property creds(*self*)
Credentials alias used by current Connection.

property current(*self*)
Returns current credentials.

property connect_kwargs(*self*) → Dict
Arguments from snowmobile.toml for *snowflake.connector.connect()*.

snowmobile.core.cfg.extensions

[external-sources] from **snowmobile.toml**.

Module Contents

Classes

Location	[external-sources]

class snowmobile.core.cfg.extensions.**Location**
Bases: snowmobile.core.cfg.base.Base

[external-sources]

ddl :Path

extensions :Path

sql_export_heading :Path

snowmobile.core.cfg.loading

[loading] section from **snowmobile.toml**, including subsections.

Module Contents

Classes

Put	[loading.put]
Copy	[loading.copy]
Loading	[loading]

class snowmobile.core.cfg.loading.**Put**
Bases: snowmobile.core.cfg.base.Base

[loading.put]

auto_compress :bool

class snowmobile.core.cfg.loading.**Copy**
Bases: snowmobile.core.cfg.base.Base

[loading.copy]

on_error :str

class snowmobile.core.cfg.loading.**Loading**
Bases: snowmobile.core.cfg.base.Base

[loading]

Default settings to use when loading data

default-file-format
Name of file-format to use when loading data into the warehouse; default is `snowmobile_default_csv`; which will be created and dropped afterwards if an existing file format is not specified;

Type str

include_index
Include the index of a DataFrame when loading it into a table; default is `False`.

Type bool

on_error
Action to take if an error is encountered when loading data into the warehouse; default is `continue`.

Type bool

keep_local
Option to keep the local file exported when loading into a staging table; default is `False`.

Type bool

include_loaded_tmstmp
Include a **loaded_tmstmp** column when loading a DataFrame into the warehouse; default is `True`.

Type bool

quote_char
Quote character to use for delimited files; default is double quotes (`"`).

Type str

auto_compress
Auto-compress file when loading data; default is `True`..

Type bool

overwrite_pre_existing_stage
Overwrite pre-existing staging table if data is being appended into an existing table/the staging table already exists; default is `True`.

Type bool

defaults :Dict

put :Put

copy_into :Copy

export_options :Dict[str, Dict]

property configured_args(*self*) → Dict
Placeholder for configuration arguments of derived classes.

snowmobile.core.cfg.script

[script] section from **snowmobile.toml**, including subsections.

Module Contents

Classes

Wildcard	[script.patterns.wildcards]
Reserved	[script.markdown.attributes.reserved]
Marker	[script.markdown.attributes.markers]
Attributes	[script.markdown.attributes]

continues on next page

Table 4 – continued from previous page

Core	[script.patterns.core]
Markup	[script.markup]
Pattern	[script.patterns]
Tolerance	[script.qa.default-tolerance]
QA	[script.qa]
Type	snowmobile-ext.toml: [wrap-to-type-xref]
Script	[script]

class snowmobile.core.cfg.script.**Wildcard**

Bases: snowmobile.core.cfg.base.Base

[script.patterns.wildcards]

char_wc :str

char_sep :str

wc_paragraph :str

wc_as_is :str

wc_omit_attr_nm :str

find_first_wc_idx(*self*, *attr_nm: str*) → int

Finds index of the first unescaped wildcard in an attribute name.

Parameters **attr_nm** (*str*) – Attribute name to search through.

Returns (int): Index position of first unescaped wildcard or 0 if one does not exist.

partition_on_wc(*self*, *attr_nm: str*) → Tuple[str, List[str]]

Parses attribute name into its display name and its wildcards.

Uses *Wildcard.find_first_wc_idx()* to determine if **attr_nm** contains a valid wildcard.

Parameters **attr_nm** (*str*) – Attribute name to parse.

Returns (Tuple[str, List[str]]): Tuple containing the attribute display name and a list of its wildcards if present and an empty list otherwise.

class snowmobile.core.cfg.script.**Reserved**

Bases: snowmobile.core.cfg.base.Base

[script.markdown.attributes.reserved]

include_by_default :bool

attr_nm :str

default_val :str

default_format :str

tabulate_format :str

class snowmobile.core.cfg.script.**Marker**(**data*)

Bases: snowmobile.core.cfg.base.Base

[script.markdown.attributes.markers]

name :str

group :str

attrs :Dict

raw :str

index :int

add(*self*, *attrs: Dict*) → *snowmobile.core.cfg.script.Marker*
Add to existing attributes.

split_attrs(*self*, *attrs: Dict*) → Tuple[Dict, Dict]
Splits attributes into user-defined-only and shared with snowmobile.toml.

Parameters **attrs** (*Dict*) – Dictionary of parsed arguments.

Returns (Tuple[Dict, Dict]): Tuple of (shared_with_snowmobile_toml_attrs, new_attrs)

update(*self*, *attrs: Dict*) → *snowmobile.core.cfg.script.Marker*
Merges parsed attributes with configuration attributes

set_name(*self*, *name: str*, *overwrite: bool = False*) → *snowmobile.core.cfg.script.Marker*
Sets the name attribute.

as_args(*self*)
Returns a dictionary of arguments for `Section`.

nm(*self*)
Marker name.

class snowmobile.core.cfg.script.**Attributes**(***data*)
Bases: snowmobile.core.cfg.base.Base

[script.markdown.attributes]

excluded :List[str]

from_namespace :Dict[str, str]

groups :Dict

order :List[str]

reserved :Dict[str, Reserved]

markers :Dict[str, Marker]

exclude(*self*, *item: str*)
Adds an item (argument name) to list of exclusions.

get_marker(*self*, *name: str*)
Fetches a template marker from *markers*.

merge_markers(*self*, *parsed_markers: Dict[int, Dict]*) → Dict[int, *Marker*]
Merges markers parsed from script with template markers in snowmobile.toml.

Does the following:

- Consumes all parsed attributes from markers found in a script
- Tries to pull a pre-configured marker based on its time and updates its pre-configured values with those provided in the script if so
- If it doesn't find a pre-configured marker based on the marker name, it will instantiate a new marker instance from the arguments provided in the script.

Parameters **parsed_markers** (*Dict[int, Dict]*) – All parsed raw marker arguments from a script by index position.

Returns (Dict[int, Marker]): Instantiated markers for all attributes, merging those with matching names to pre-configured markers in snowmobile.toml.

get_position (*self*, *attr: str*) → int
Returns the position for an attribute based on snowmobile-ext.toml.

Will return 0 if not included in order-by configuration.

included (*self*, *attrs: Dict*) → Dict
Checks if an attribute has been marked for exclusion from render.

group_parsed_attrs (*self*, *parsed: Dict*) → Dict
Nests attributes into dictionaries that are configured as groups.

add_reserved_attrs (*self*, *attrs: Dict*, *is_marker: bool = False*)
Batch modifies all reserved attributes to their configuration.

class snowmobile.core.cfg.script.**Core**
Bases: snowmobile.core.cfg.base.Base

[script.patterns.core]

to_open :str

to_close :str

delimiter :str

prefix :str

class snowmobile.core.cfg.script.**Markup**
Bases: snowmobile.core.cfg.base.Base

[script.markup]

hx_marker :str

hx_statement :str

bullet_char :str

attr_nm_wrap_char :str

attr_value_wrap_char :str

attrs :Attributes

pref_header (*self*, *is_marker: bool = False*, *from_wc: Optional[str] = False*) → str
Creates header prefix based on specifications.

class snowmobile.core.cfg.script.**Pattern**
Bases: snowmobile.core.cfg.base.Base

[script.patterns]

core :Core

wildcards :Wildcard

class snowmobile.core.cfg.script.**Tolerance**
Bases: snowmobile.core.cfg.base.Base

[script.qa.default-tolerance]

relative :float

absolute :float

only_matching_rows :bool

class snowmobile.core.cfg.script.**QA**
Bases: snowmobile.core.cfg.base.Base

[script.qa]

partition_on :str

ignore_patterns :List

compare_patterns :List

tolerance :Tolerance

class snowmobile.core.cfg.script.**Type**
Bases: snowmobile.core.cfg.base.Base

snowmobile-ext.toml: [wrap-to-type-xref]

as_str :List

as_list :List

as_float :List

as_bool :List

class snowmobile.core.cfg.script.**Script**
Bases: snowmobile.core.cfg.base.Base

[script]

patterns :Pattern

markup :Markup

qa :QA

types :Type

export_dir_nm :str

result_limit :int

tag(*self*) → Tuple[str, str]
Open/close pattern for statement tags.

static power_strip(*val_to_strip: str*, *chars_to_strip: Iterable[str]*) → str
Exhaustively strips a string of specified leading/trailing characters.

arg_to_string(*self*, *arg_as_str: str*) → str
Strips an argument as a string down to its elemental form.

arg_to_list(*self*, *arg_as_str: str*) → List[str]
Converts a list as a string into a list.

arg_to_float(*self*, *arg_as_str: str*) → float
Strips a string down to its elemental form and converts to a float.

arg_to_bool(*self*, *arg_as_str: str*) → bool
Converts a boolean in string-form into a boolean value.

parse_arg (*self*, *arg_key: str*, *arg_value: str*)
Parses an argument into its target data type based on its *arg_key* and the `script.name-to-type-xref` defined in **snowmobile.toml**.

static split_args (*args_str: str*) → List[str]
Returns a list of arguments based on splitting string on double underscores and stripping results.

parse_split_arguments (*self*, *splitter: List[str]*) → Dict
Returns a dictionary of argument-index to argument keys and values.

parse_str (*self*, *block: str*, *strip_blanks: bool = False*, *strip_trailing: bool = False*) → Dict
Parses a string of statement tags/arguments into a valid dictionary.

Parameters

- **block** (*str*) – Raw string of all text found between a given open/close wrap.
- **strip_blanks** (*bool*) – Strip blank lines from string; defaults to *False*.
- **strip_trailing** (*bool*) – Strip trailing whitespace from the string; defaults to *False*.

Returns (dict): Dictionary of arguments.

wrap (*self*, *tag: str*) → str
Wraps a raw string of sql in open/closing patterns.

static attr_construct (*attr_nm: str*, *attr_value: str*) → str
Returns a parsable attribute from an attribute name and value.

tag_from_attrs (*self*, *attrs: Dict*, *nm: Optional[str] = None*, *wrap: bool = False*, ***kwargs*) → str
Construct a parsable tag out of a dictionary of attributes.

as_parsable (*self*, *raw: str*, *is_multiline: Optional[bool] = None*, *is_marker: Optional[bool] = None*, *lines: Optional[int] = None*) → str
Returns a raw string wrapped in open/close tags.

Used for taking a raw string of marker or statement attributes and wrapping it in open/close tags before exporting, making the script being exported re-parsable by *snowmobile*.

find_spans (*self*, *sql: str*) → Dict[int, Tuple[int, int]]
Finds indices of script tags given a sql script as a string and an open and close pattern of the tags.

find_tags (*self*, *sql: str*) → Dict[int, str]
Finds indices of script tags given a sql script as a string and an open and close pattern of the tags.

find_block (*self*, *sql: str*, *marker: str*) → str
Finds a block of arguments based on a marker.

Markers expected by default are the __script__ and __appendix__ markers.

has_tag (*self*, *s: sqlparse.sql.Statement*) → bool
Checks if a given statement has a wrap that directly precedes the sql.

static is_marker (*raw: str*)
Checks if a raw string of arguments has a marker on the first line.

static is_valid_sql (*s: sqlparse.sql.Statement*) → bool
Verifies that a given `sqlparse.sql.Statement` contains valid sql.

static strip_comments (*s: sqlparse.sql.Statement*) → str
Isolates just the sql within a `sqlparse.sql.Statement` object.

split_sub_blocks (*self*, *s: sqlparse.sql.Statement*) → Tuple[List, str]
Breaks apart blocks of arguments within a `sqlparse.sql.Statement`.

Note:

- `sqlparse.parsestream()` returns a list of `sqlparse.sql.Statement` objects, each of which includes all text (comments) between the last character of the prior statement and the first character of the current one.
- *split_sub_blocks()* traverses that space and identifies all spans of text wrapped in *open* (`/*-`) and *close* (`-*/`) tags, storing their index positions relative to the other statements & markers.
- These are stored as *snowmobile.core.Script* attributes as statements are parsed and so that they can be exported in the appropriate order to a markdown file.

Parameters **s** (*sqlparse.sql.Statement*) – `sqlparse.sql.Statement` object.

Returns (Tuple[List, str]):

A tuple containing:

1. A list of __marker__ blocks if they exist; empty list otherwise
2. The last wrap/block before the start of the actual SQL (e.g. the wrap/block that is associated with the statement passed to `s`.

name_from_marker (*self*, *raw: str*) → str
Extracts a marker name (e.g. 'script' from within __script__).

parse_name (*self*, *raw: str*, *offset: Optional[int] = None*, *silence: bool = False*) → str
Parses name from a raw set of arguments if not given an explicit wrap.

static add_name (*nm_title: str*, *nm_marker: str*, *attrs: dict*, *overwrite: bool = False*)
Adds a name to a set of parsed marker attributes.

Accepts a name and a dict of parsed attributes from a marker and:

1. Checks to see if there's an explicit 'name' declared within the attributes
2. If not explicitely declared **or** explicitely declared and *overwrite=False*, it will add the *nm* value to the attributes as 'name'.
3. It will also add the 'nm' value to the attributes as 'marker-name' to be used by the *Marker* when cross-referencing the __name__ with template markers in `snowmobile.toml`.

Parameters

- **nm_title** (*str*) –

  The name of the marker as either:

  1. Returned value from *name_from_marker()*
  2. Returned value from *parse_name()*
  3. None if neither is provided nm_marker (str):

  The string value wrapped in ___ on the first line of the argument block.

- **attrs** (*dict*) – A dictionary of parsed attributes as returned from *parse_str()*.

- **overwrite** (*bool*) – Indicator of whether or not to overwrite a 'name' attribute declared within the .sql script.

parse_marker (*self*, *attrs_raw: str*) → Dict
Parses a raw string of __marker__ text between an open and a close pattern.

static ensure_sqlparse (*sql: Union[sqlparse.sql.Statement, str]*) → sqlparse.sql.Statement
Returns a `sqlparse.sql.Statement` from `sql`.

Will return `sql` with no modification if it's already a `sqlparse.sql` object.

Needed to accommodate dynamic addition of statements as strings to an existing `Script` object from from raw strings as opposed to a `sqlparse.sql.Statement` objects as is done when reading a sql file.

Parameters **sql** (*Union[sqlparse.sql.Statement, str]*) – Either a string of sql or an already parsed sqlparse.sql.Statement object.

Returns (sqlparse.sql.Statement): A parsed sql statement.

sql_tokens (*self*, *sql: str*) → List[sqlparse.sql.Token]
Unpacks nested tokens from a sqlparse.sql.Statement.

Parameters **sql** (*str*) – A raw sql from a statement.

Returns A list of tokens.

id_from_tokens (*self*, *sql: str*) → str
Identifies the last identifier in a piece of raw sql.

Identifies *obj* being operated on.

Parameters **sql** (*str*) – A raw piece of sql from a statement.

Returns A string if the last identifier found or an empty string otherwise.

snowmobile.core.cfg.sql

[sql] (**snowmobile-ext.toml**)

Module Contents

Classes

SQL	[sql] (**snowmobile-ext.toml**)

class snowmobile.core.cfg.sql.**SQL**
Bases: `snowmobile.core.cfg.base.Base`

[sql] (**snowmobile-ext.toml**)

generic_anchors :Dict

kw_exceptions :Dict

named_objects :List

info_schema_exceptions :Dict[str, str]

desc_is_simple :bool

pr_over_ge :bool

info_schema_loc(*self*, *obj: str*, *stem: bool = False*) → str
Returns information schema table for object if other than making plural.

i.e.:

- 'tables' -> 'tables'
- 'table' –> 'tables'
- 'schemas' -> 'schemata'
- 'schema' –> 'schemata'

objects_within(*self*, *first_line: str*)
Searches the first line of sql for matches to named objects.

Package Contents

Classes

Base	Base class for object model parsed from `snowmobile.toml`.
Connection	[connection]
Credentials	[connection.credentials.credentials_alias]
Loading	[loading]
Put	[loading.put]
Copy	[loading.copy]
SQL	[sql] (**snowmobile-ext.toml**)
Location	[external-sources]
QA	[script.qa]
Attributes	[script.markdown.attributes]
Markup	[script.markup]
Marker	[script.markdown.attributes.markers]
Pattern	[script.patterns]
Script	[script]
Wildcard	[script.patterns.wildcards]

class snowmobile.core.cfg.**Base**
Bases: `pydantic.BaseModel`, `snowmobile.core.cfg.base.Config`

Base class for object model parsed from `snowmobile.toml`.

property configured_args(*self*) → Dict
Placeholder for configuration arguments of derived classes.

kwarg(*self*, *arg_nm: str*, *arg_val: Any*, *arg_typ: Any*) → Any
Compares a provided keyword argument to a configured keyword argument.

from_relative(*self*, *obj: Any*)
Updates current object's attributes with those from a different instance of the same class.

from_dict(*self*, *args: Dict*)
Accept a dictionary of arguments and updates the current object as if it were instantiated with those arguments.

as_serializable(*self*, *by_alias: bool = False*)
Returns a dictionary in serializable form.

json(*self*, *by_alias: bool = False*, ***kwargs*) → str
API-facing json serialization method.

class snowmobile.core.cfg.**Connection**(***data*)
Bases: snowmobile.core.cfg.base.Base

[connection]

This includes the *default_alias* which is the set of credentials that snowmobile will authenticate with if *creds* is not explicitly passed.

default_alias
The set of credentials that is used if *creds* is not explicitly passed to snowmobile.connect on instantiation.

Type str

creds
The name given to the set of credentials within the **credentials** block of the **snowmobile.toml** file (e.g. [credentials.creds] assigns an *creds* to a given set of credentials.

Type str

creds
A dictionary of *creds* to the associated Creds object containing its credentials.

Type dict[str, Creds]

default_alias :str

provided_alias :str

credentials :Dict[str, Credentials]

defaults :Dict

property creds(*self*)
Credentials alias used by current Connection.

property current(*self*)
Returns current credentials.

property connect_kwargs(*self*) → Dict
Arguments from snowmobile.toml for *snowflake.connector.connect()*.

class snowmobile.core.cfg.**Credentials**
Bases: snowmobile.core.cfg.base.Base

[connection.credentials.credentials_alias]

user :str

password :str

role :str

account :str

warehouse :str

database :str

schema_name :str

as_nm(*self*, *n: str*)
Sets the credentials alias.

property credentials(*self*)
Returns namespace as a dictionary, excluding `_alias`.

class snowmobile.core.cfg.**Loading**
Bases: snowmobile.core.cfg.base.Base

[loading]

Default settings to use when loading data

default-file-format
Name of file-format to use when loading data into the warehouse; default is `snowmobile_default_csv`; which will be created and dropped afterwards if an existing file format is not specified;

Type str

include_index
Include the index of a DataFrame when loading it into a table; default is `False`.

Type bool

on_error
Action to take if an error is encountered when loading data into the warehouse; default is `continue`.

Type bool

keep_local
Option to keep the local file exported when loading into a staging table; default is `False`.

Type bool

include_loaded_tmstmp
Include a **loaded_tmstmp** column when loading a DataFrame into the warehouse; default is `True`.

Type bool

quote_char
Quote character to use for delimited files; default is double quotes (`"`).

Type str

auto_compress
Auto-compress file when loading data; default is `True`..

Type bool

overwrite_pre_existing_stage
Overwrite pre-existing staging table if data is being appended into an existing table/the staging table already exists; default is `True`.

Type bool

defaults :Dict

put :Put

copy_into :Copy

export_options :Dict[str, Dict]

property configured_args(*self*) → Dict
Placeholder for configuration arguments of derived classes.

class snowmobile.core.cfg.**Put**
Bases: snowmobile.core.cfg.base.Base

[loading.put]

auto_compress :bool

class snowmobile.core.cfg.**Copy**
Bases: snowmobile.core.cfg.base.Base

[loading.copy]

on_error :str

class snowmobile.core.cfg.**SQL**
Bases: snowmobile.core.cfg.base.Base

[sql] (**snowmobile-ext.toml**)

generic_anchors :Dict

kw_exceptions :Dict

named_objects :List

info_schema_exceptions :Dict[str, str]

desc_is_simple :bool

pr_over_ge :bool

info_schema_loc(*self*, *obj: str*, *stem: bool = False*) → str
Returns information schema table for object if other than making plural.

i.e.:

- 'tables' -> 'tables'
- 'table' –> 'tables'
- 'schemas' -> 'schemata'
- 'schema' –> 'schemata'

objects_within(*self*, *first_line: str*)
Searches the first line of sql for matches to named objects.

class snowmobile.core.cfg.**Location**
Bases: snowmobile.core.cfg.base.Base

[external-sources]

ddl :Path

extensions :Path

sql_export_heading :Path

class snowmobile.core.cfg.**QA**
Bases: snowmobile.core.cfg.base.Base

[script.qa]

partition_on :str

ignore_patterns :List

compare_patterns :List

tolerance :Tolerance

class snowmobile.core.cfg.**Attributes**(***data*)
Bases: snowmobile.core.cfg.base.Base

[script.markdown.attributes]

excluded :List[str]

from_namespace :Dict[str, str]

groups :Dict

order :List[str]

reserved :Dict[str, Reserved]

markers :Dict[str, Marker]

exclude(*self*, *item: str*)
Adds an item (argument name) to list of exclusions.

get_marker(*self*, *name: str*)
Fetches a template marker from *markers*.

merge_markers(*self*, *parsed_markers: Dict[int, Dict]*) → Dict[int, *Marker*]
Merges markers parsed from script with template markers in snowmobile.toml.

Does the following:

- Consumes all parsed attributes from markers found in a script
- Tries to pull a pre-configured marker based on its time and updates its pre-configured values with those provided in the script if so
- If it doesn't find a pre-configured marker based on the marker name, it will instantiate a new marker instance from the arguments provided in the script.

Parameters **parsed_markers** (*Dict[int, Dict]*) – All parsed raw marker arguments from a script by index position.

Returns (Dict[int, Marker]): Instantiated markers for all attributes, merging those with matching names to pre-configured markers in snowmobile.toml.

get_position(*self*, *attr: str*) → int
Returns the position for an attribute based on snowmobile-ext.toml.

Will return 0 if not included in order-by configuration.

included(*self*, *attrs: Dict*) → Dict
Checks if an attribute has been marked for exclusion from render.

group_parsed_attrs(*self*, *parsed: Dict*) → Dict
Nests attributes into dictionaries that are configured as groups.

add_reserved_attrs(*self*, *attrs: Dict*, *is_marker: bool = False*)
Batch modifies all reserved attributes to their configuration.

class snowmobile.core.cfg.**Markup**
Bases: snowmobile.core.cfg.base.Base

[script.markup]

hx_marker :str

hx_statement :str

bullet_char :str

attr_nm_wrap_char :str

attr_value_wrap_char :str

attrs :Attributes

pref_header(*self*, *is_marker: bool = False*, *from_wc: Optional[str] = False*) → str
Creates header prefix based on specifications.

class snowmobile.core.cfg.**Marker**(***data*)
Bases: snowmobile.core.cfg.base.Base

[script.markdown.attributes.markers]

name :str

group :str

attrs :Dict

raw :str

index :int

add(*self*, *attrs: Dict*) → *snowmobile.core.cfg.script.Marker*
Add to existing attributes.

split_attrs(*self*, *attrs: Dict*) → Tuple[Dict, Dict]
Splits attributes into user-defined-only and shared with snowmobile.toml.

Parameters **attrs** (*Dict*) – Dictionary of parsed arguments.

Returns (Tuple[Dict, Dict]): Tuple of (shared_with_snowmobile_toml_attrs, new_attrs)

update(*self*, *attrs: Dict*) → *snowmobile.core.cfg.script.Marker*
Merges parsed attributes with configuration attributes

set_name(*self*, *name: str*, *overwrite: bool = False*) → *snowmobile.core.cfg.script.Marker*
Sets the name attribute.

as_args(*self*)
Returns a dictionary of arguments for Section.

nm(*self*)
Marker name.

class snowmobile.core.cfg.**Pattern**
Bases: snowmobile.core.cfg.base.Base

[script.patterns]

core :Core

wildcards :Wildcard

class snowmobile.core.cfg.**Script**
Bases: snowmobile.core.cfg.base.Base

[script]

patterns :Pattern

markup :Markup

qa :QA

types :Type

export_dir_nm :str

result_limit :int

tag (*self*) → Tuple[str, str]
Open/close pattern for statement tags.

static power_strip (*val_to_strip: str*, *chars_to_strip: Iterable[str]*) → str
Exhaustively strips a string of specified leading/trailing characters.

arg_to_string (*self*, *arg_as_str: str*) → str
Strips an argument as a string down to its elemental form.

arg_to_list (*self*, *arg_as_str: str*) → List[str]
Converts a list as a string into a list.

arg_to_float (*self*, *arg_as_str: str*) → float
Strips a string down to its elemental form and converts to a float.

arg_to_bool (*self*, *arg_as_str: str*) → bool
Converts a boolean in string-form into a boolean value.

parse_arg (*self*, *arg_key: str*, *arg_value: str*)
Parses an argument into its target data type based on its *arg_key* and the `script.name-to-type-xref` defined in **snowmobile.toml**.

static split_args (*args_str: str*) → List[str]
Returns a list of arguments based on splitting string on double underscores and stripping results.

parse_split_arguments (*self*, *splitter: List[str]*) → Dict
Returns a dictionary of argument-index to argument keys and values.

parse_str (*self*, *block: str*, *strip_blanks: bool = False*, *strip_trailing: bool = False*) → Dict
Parses a string of statement tags/arguments into a valid dictionary.

Parameters

- **block** (*str*) – Raw string of all text found between a given open/close wrap.
- **strip_blanks** (*bool*) – Strip blank lines from string; defaults to *False*.
- **strip_trailing** (*bool*) – Strip trailing whitespace from the string; defaults to *False*.

Returns (dict): Dictionary of arguments.

wrap (*self*, *tag: str*) → str
Wraps a raw string of sql in open/closing patterns.

static attr_construct (*attr_nm: str*, *attr_value: str*) → str
Returns a parsable attribute from an attribute name and value.

tag_from_attrs (*self*, *attrs: Dict*, *nm: Optional[str] = None*, *wrap: bool = False*, ***kwargs*) → str
Construct a parsable tag out of a dictionary of attributes.

as_parsable (*self*, *raw: str*, *is_multiline: Optional[bool] = None*, *is_marker: Optional[bool] = None*, *lines: Optional[int] = None*) → str
Returns a raw string wrapped in open/close tags.

Used for taking a raw string of marker or statement attributes and wrapping it in open/close tags before exporting, making the script being exported re-parsable by *snowmobile*.

find_spans (*self*, *sql: str*) → Dict[int, Tuple[int, int]]
Finds indices of script tags given a sql script as a string and an open and close pattern of the tags.

find_tags (*self*, *sql: str*) → Dict[int, str]
Finds indices of script tags given a sql script as a string and an open and close pattern of the tags.

find_block (*self*, *sql: str*, *marker: str*) → str
Finds a block of arguments based on a marker.

Markers expected by default are the __script__ and __appendix__ markers.

has_tag (*self*, *s: sqlparse.sql.Statement*) → bool
Checks if a given statement has a wrap that directly precedes the sql.

static is_marker (*raw: str*)
Checks if a raw string of arguments has a marker on the first line.

static is_valid_sql (*s: sqlparse.sql.Statement*) → bool
Verifies that a given `sqlparse.sql.Statement` contains valid sql.

static strip_comments (*s: sqlparse.sql.Statement*) → str
Isolates just the sql within a `sqlparse.sql.Statement` object.

split_sub_blocks (*self*, *s: sqlparse.sql.Statement*) → Tuple[List, str]
Breaks apart blocks of arguments within a `sqlparse.sql.Statement`.

Note:

- `sqlparse.parsestream()` returns a list of `sqlparse.sql.Statement` objects, each of which includes all text (comments) between the last character of the prior statement and the first character of the current one.
- *`split_sub_blocks()`* traverses that space and identifies all spans of text wrapped in *open* (`/*-`) and *close* (`-*/`) tags, storing their index positions relative to the other statements & markers.
- These are stored as *`snowmobile.core.Script`* attributes as statements are parsed and so that they can be exported in the appropriate order to a markdown file.

Parameters **s** (*`sqlparse.sql.Statement`*) – `sqlparse.sql.Statement` object.

Returns (Tuple[List, str]):

A tuple containing:

1. A list of __marker__ blocks if they exist; empty list otherwise
2. The last wrap/block before the start of the actual SQL (e.g. the wrap/block that is associated with the statement passed to `s`.

name_from_marker (*self*, *raw: str*) → str
Extracts a marker name (e.g. 'script' from within __script__).

parse_name (*self*, *raw: str*, *offset: Optional[int] = None*, *silence: bool = False*) → str
Parses name from a raw set of arguments if not given an explicit wrap.

static add_name (*nm_title: str*, *nm_marker: str*, *attrs: dict*, *overwrite: bool = False*)
Adds a name to a set of parsed marker attributes.

Accepts a name and a dict of parsed attributes from a marker and:

1. Checks to see if there's an explicit 'name' declared within the attributes

2. If not explicitely declared **or** explicitely declared and *overwrite=False*, it will add the *nm* value to the attributes as 'name'.
3. It will also add the 'nm' value to the attributes as 'marker-name' to be used by the *Marker* when cross-referencing the __name__ with template markers in `snowmobile.toml`.

Parameters

- **nm_title** (*str*) –

  The name of the marker as either:

  1. Returned value from *name_from_marker()*
  2. Returned value from *parse_name()*
  3. None if neither is provided nm_marker (str):

  The string value wrapped in ___ on the first line of the argument block.

- **attrs** (*dict*) – A dictionary of parsed attributes as returned from *parse_str()*.
- **overwrite** (*bool*) – Indicator of whether or not to overwrite a 'name' attribute declared within the .sql script.

parse_marker (*self*, *attrs_raw: str*) → Dict
Parses a raw string of __marker__ text between an open and a close pattern.

static ensure_sqlparse (*sql: Union[sqlparse.sql.Statement, str]*) → sqlparse.sql.Statement
Returns a `sqlparse.sql.Statement` from `sql`.

Will return `sql` with no modification if it's already a `sqlparse.sql` object.

Needed to accommodate dynamic addition of statements as strings to an existing `Script` object from from raw strings as opposed to a `sqlparse.sql.Statement` objects as is done when reading a sql file.

Parameters **sql** (*Union[sqlparse.sql.Statement, str]*) – Either a string of sql or an already parsed sqlparse.sql.Statement object.

Returns (sqlparse.sql.Statement): A parsed sql statement.

sql_tokens (*self*, *sql: str*) → List[sqlparse.sql.Token]
Unpacks nested tokens from a sqlparse.sql.Statement.

Parameters **sql** (*str*) – A raw sql from a statement.

Returns A list of tokens.

id_from_tokens (*self*, *sql: str*) → str
Identifies the last identifier in a piece of raw sql.

Identifies *obj* being operated on.

Parameters **sql** (*str*) – A raw piece of sql from a statement.

Returns A string if the last identifier found or an empty string otherwise.

class snowmobile.core.cfg.**Wildcard**
Bases: `snowmobile.core.cfg.base.Base`

[script.patterns.wildcards]

char_wc :str

char_sep :str

wc_paragraph :str

wc_as_is :str

wc_omit_attr_nm :str

find_first_wc_idx(*self*, *attr_nm: str*) → int
Finds index of the first unescaped wildcard in an attribute name.

Parameters **attr_nm** (*str*) – Attribute name to search through.

Returns (int): Index position of first unescaped wildcard or 0 if one does not exist.

partition_on_wc(*self*, *attr_nm: str*) → Tuple[str, List[str]]
Parses attribute name into its display name and its wildcards.

Uses *Wildcard.find_first_wc_idx()* to determine if **attr_nm** contains a valid wildcard.

Parameters **attr_nm** (*str*) – Attribute name to parse.

Returns (Tuple[str, List[str]]): Tuple containing the attribute display name and a list of its wildcards if present and an empty list otherwise.

7.2 Submodules

7.2.1 snowmobile.core.configuration

snowmobile.core.Configuration is a parsed snowmobile.toml file; class handles:

1. Locating *snowmobile.toml*, from:
   a. A cached location specific to the version of snowmobile and the file name (defaults to *snowmobile.toml*)
   b. Finding a file based on its name from traversing the file system, used when initially finding snowmobile.toml or when a bespoke configuration file name has been provided
2. Checking **[ext-sources]** for specified external configurations
3. Instantiating each section in snowmobile.toml from the (Pandantic) models defined in *snowmobile.core.cfg*; root sections are set as individual attributes on *Configuration*

Module Contents

Classes

Configuration	A parsed *snowmobile.toml* file.

class snowmobile.core.configuration.**Configuration**(*creds: Optional[str] = None*, *config_file_nm: Optional[str] = None*, *from_config: Optional[Path, str] = None*, *export_dir: Optional[Path, str] = None*, *silence: bool = False*)

Bases: snowmobile.core.base.Generic

A parsed *snowmobile.toml* file.

All keyword arguments optional.

Parameters

- **config_file_nm** (*Optional[str]*) – Name of configuration file to use; defaults to *snowmobile.toml*.
- **creds** (*Optional[str]*) – Alias for the set of credentials to authenticate with; default behavior will fall back to the *connection.default-creds* specified in *snowmobile.toml*, *or the first set of credentials stored if this configuration option is left blank*.
- **from_config** (*Optional[str, Path]*) – A full path to a specific configuration file to use; bypasses any checks for a cached file location and can be useful for container-based processes with restricted access to the local file system.
- **export_dir** (*Optional[Path]*) – Path to save a template *snowmobile.toml* file to; if pr, the file will be exported within the __init__ method and nothing else will be instantiated.

file_nm
Configuration file name; defaults to 'snowmobile.toml'.

Type str

cache
Persistent cache; caches *location*.

Type snowmobile.core.cache.Cache

location
Full path to configuration file.

Type pathlib.Path

connection :Optional[cfg.Connection]
[connection] from snowmobile.toml.

Type *snowmobile.core.cfg.Connection*

loading :Optional[cfg.Loading]
[loading] from snowmobile.toml.

Type *snowmobile.core.cfg.Loading*

script :Optional[cfg.Script]
[script] from snowmobile.toml.

Type *snowmobile.core.cfg.Script*

sql :Optional[cfg.SQL]
[sql] from snowmobile-ext.toml.

Type *snowmobile.core.cfg.SQL*

ext_sources :Optional[cfg.Location]
[external-sources] from snowmobile.toml.

Type *snowmobile.core.cfg.Location*

property markdown(*self*) → *snowmobile.core.cfg.Markup*
Accessor for cfg.script.markdown.

property attrs(*self*) → *snowmobile.core.cfg.Attributes*
Accessor for cfg.script.markdown.attributes.

property wildcards (*self*) → *snowmobile.core.cfg.Wildcard*
Accessor for cfg.script.patterns.wildcards.

static batch_set_attrs (*obj: Any*, *attrs: dict*, *to_none: bool = False*)
Batch sets attributes on an object from a dictionary.

Parameters

- **obj** (*Any*) – Object to set attributes on.
- **attrs** (*dict*) – Dictionary containing attributes.
- **to_none** (*bool*) – Set all of the object's attributes batching a key in *wrap* to *None*; defaults ot *False*.

Returns (Any): Object post-setting attributes.

static attrs_from_obj (*obj: Any*, *within: Optional[List[str]] = None*) → Dict[str, MethodType]
Utility to return attributes/properties from an object as a dictionary.

static methods_from_obj (*obj: Any*, *within: Optional[List[str]] = None*) → Dict[str, MethodType]
Returns callable components of an object as a dictionary.

property scopes (*self*)
All combinations of scope type and scope attribute.

scopes_from_kwargs (*self*, *only_populated: bool = False*, ***kwargs*) → Dict
Turns *script.filter()* arguments into a valid set of kwargs for `Scope`.

Returns dictionary of all combinations of 'arg' ("kw", "obj", "desc", "anchor" and "nm"), including empty sets for any 'arg' not included in the keyword arguments provided.

scopes_from_tag (*self*, *t: Any*)
Generates list of keyword arguments to instantiate all scopes for a wrap.

json (*self*, *by_alias: bool = False*, ***kwargs*)
Serialization method for core object model.

7.2.2 snowmobile.core.connection

Snowmobile is the core of snowmobile's object model and a given instance is often shared across multiple objects at once.

It is the primary method of executing statement against the warehouse and it stores the fully parsed & validated `snowmobile.toml` file it was instantiated with as its *cfg* attribute.

Within `snowmobile`'s code and documentation, it is referred to as `sn` for brevity.

Module Contents

Classes

Snowmobile	Primary method of statement execution and accessor to parsed snowmobile.toml.

class snowmobile.core.connection.**Snowmobile**(*creds: Optional[str] = None*, *delay: bool = False*, *ensure_alive: bool = True*, *config_file_nm: Optional[str] = None*, *from_config: Optional[str, Path] = None*, *silence: bool = False*, ***connect_kwargs*)

Bases: *snowmobile.core.sql.SQL*

Primary method of statement execution and accessor to parsed snowmobile.toml.

Parameters

- **creds** (*Optional[str]*) – Alias for the set of credentials to authenticate with; default behavior will fall back to the `connection.default-creds` specified in *snowmobile.toml*, *or the first set of credentials stored if this configuration option is left blank*.
- **delay** (*bool*) – Optionally delay establishing a connection when the object is instantiated, enabling access to the configuration object model through the `Connection.cfg` attribute; defaults to *False*.
- **ensure_alive** (*bool*) – Establish a new connection if a method requiring a connection against the database is called while *alive* is *False*; defaults to *True*.
- **config_file_nm** (*Optional[str]*) – Name of configuration file to use; defaults to *snowmobile.toml*.
- **from_config** (*Optional[str, Path]*) – A full path to a specific configuration file to use; bypasses any checks for a cached file location and can be useful for container-based processes with restricted access to the local file system.
- ****connect_kwargs** – Additional arguments to provide to snowflake.connector.connect(); arguments provided here will over-ride connection arguments specified in *snowmobile.toml*, including:
  - Connection parameters in *connection.default-arguments*
  - Credentials parameters associated with a given alias
  - Connection parameters associated with a given alias

Initializes a `snowmobile.SQL` object.

cfg :Configuration
snowmobile.toml

Type *snowmobile.core.configuration.Configuration*

con :Optional[SnowflakeConnection]
Can be *None* until set by *Snowmobile.connect()*

Type SnowflakeConnection

e :ExceptionHandler
Exception / context management

Type snowmobile.core.exception_handler.ExceptionHandler

ensure_alive :bool
Reconnect to Snowflake if connection is lost

Type bool

connect(*self*, ***kwargs*) → *snowmobile.core.connection.Snowmobile*
Establishes connection to Snowflake.

Re-implements *snowflake.connector.connect()* with connection arguments sourced from snowmobile's object model, specifically:

- Credentials from *snowmobile.toml*.
- Default connection arguments from *snowmobile.toml*.
- Optional keyword arguments either passed to `snowmobile.connect()` or directly to this method.

kwargs: Optional keyword arguments to pass to snowflake.connector.connect(); arguments passed here will over-ride `connection.default-arguments` specified in `snowmobile.toml`.

disconnect(*self*) → *snowmobile.core.connection.Snowmobile*
Disconnect from connection with which Connection() was instantiated.

property alive(*self*) → bool
Check if the connection is alive.

property cursor(*self*) → snowflake.connector.connection.SnowflakeCursor
`SnowflakeCursor` accessor.

property dictcursor(*self*) → snowflake.connector.DictCursor
`DictCursor` accessor.

ex(*self*, *sql: str*, *on_error: Optional[str] = None*, ***kwargs*) → snowflake.connector.connection.SnowflakeCursor
Executes a command via `SnowflakeCursor`.

Parameters

- **sql** (*str*) – `sql` command as a string.
- **on_error** (*str*) – String value to impose a specific behavior if an error occurs during the execution of `sql`.
- ****kwargs** – Optional keyword arguments for `SnowflakeCursor.execute()`.

Returns (SnowflakeCursor): `SnowflakeCursor` object that executed the command.

exd(*self*, *sql: str*, *on_error: Optional[str] = None*, ***kwargs*) → snowflake.connector.DictCursor
Executes a command via `DictCursor`.

Parameters

- **sql** (*str*) – `sql` command as a string.
- **on_error** (*str*) – String value to impose a specific behavior if an error occurs during the execution of `sql`.
- ****kwargs** – Optional keyword arguments for `SnowflakeCursor.execute()`.

Returns (DictCursor): `DictCursor` object that executed the command.

query(*self*, *sql: str*, *as_df: bool = False*, *as_cur: bool = False*, *as_dcur: bool = False*, *as_scalar: bool = False*, *lower: bool = True*, *on_error: Optional[str] = None*) → Union[pd.DataFrame, SnowflakeCursor]
Execute a query and return results.

Default behavior of *results=True* will return results as a `pandas.DataFrame`, otherwise will execute the sql provided with a `SnowflakeCursor` and return the cursor object.

Parameters

- **sql** (*str*) – Raw SQL to execute.
- **as_df** (*bool*) – Return results in DataFrame.
- **as_cur** (*bool*) – Return results in Cursor.
- **as_dcur** (*bool*) – Return results in a DictCursor.
- **as_scalar** (*bool*) – Return results as a single scalar value.
- **lower** (*bool*) – Boolean value indicating whether or not to return results with columns lower-cased.
- **on_error** (*str*) – String value to impose a specific behavior if an error occurs during the execution of `sql`.

Returns (Union[pd.DataFrame, SnowflakeCursor]): Results from `sql` as a `DataFrame` by default or the `SnowflakeCursor` object if *results=False*.

7.2.3 snowmobile.core.markup

Calling *script.doc()* returns a *Markup* containing a `Section` for each statement or marker within the script.

Note: A *Markup* instance, `m`, returned by *script.doc()*, makes no modifications to the sql file read by `script`

Instead, `m` will generate and export the following two files:

- A sql file stripped of all untagged comments, limited to statements within the context of `script` at the time `m` was created
- A markdown representation of the code and markup associated with the same set of statements

By default, these files are exported to a `.snowmobile` directory alongside the sql file that was read by the `script`; the directory name to use for generated exports can be configured in *[script.export-dir-name]*

If the target directory does not yet exist, it will be created as part of the export process invoked by *m.save()*

The *default markdown configuration* yields a *.md* file with the below structure:

```
# Script Name.sql          [script name gets an 'h1' header]

- **Key1**: *Value1*       [keys are bolded, values are italicized]
- **Key2**: *Value2*       [same for all tags/attributes found]
- ...

**Description**            [except for the 'Description' section]
                           [this is just a blank canvas of markdown..]
                           [..but this is configurable]

## (1) create-table~dummy_name [contents get 'h2' level headers]

- **Key1**: *Value1*       [identical formatting for st/markers]

**Description**            [statement descriptions get one of these too]

**SQL**                    [as does their rendered sql]
    ...sql
```

(continues on next page)

(continued from previous page)

```
        ...
        ...
    ...


## (2) update-table~dummy_name2

...                         [structure repeats for all contents in the script]
```

Module Contents

Classes

Markup	Contains all sections within the context of a `Script`.

class snowmobile.core.markup.**Markup**(*sn:* snowmobile.core.connection.Snowmobile, *path:* *pathlib.Path*, *contents:* *Dict[int, Union[Statement, Marker]]*, *nm:* *Optional[str]* = *None*, *prefix:* *Optional[str]* = *None*, *suffix:* *Optional[str]* = *None*, *root_dir:* *Optional[Union[str, Path]]* = *None*, *sub_dir:* *Optional[str]* = *None*, *incl_sql:* *bool* = *True*, *incl_markers:* *bool* = *True*, *incl_sql_tag:* *bool* = *False*, *incl_exp_ctx:* *bool* = *True*, *result_wrap:* *Optional[str]* = *None*)

Bases: *snowmobile.core.Generic*

Contains all sections within the context of a `Script`.

Parameters

- **sn** (*Snowmobile*) – A `Snowmobile` instance.
- **path** (*Path*) – A full path to the sql file that script was instantiated from.
- **contents** (*Dict[int, Union[Statement, Marker]]*) – A dictionary of the script's contents (st and markers) by index position.
- **nm** (*Optional[str]*) – Alternate file name to use; defaults to `path.name`.
- **prefix** (*Optional[str]*) – Prefix to prepend to original file name when exporting.
- **suffix** (*Optional[str]*) – Suffix to append to original file name when exporting.
- **root_dir** (*Optional[Union[str, Path]]*) – Alternate target directory for exports; defaults to `./.snowmobile` where `.` is the directory containing the sql file that the `script` was created from.
- **sub_dir** (*Optional[str]*) – Alternate sub-directory name; defaults to `path.name` where `path` is a full `Path` to the sql file that the `script` was created from.
- **incl_sql** (*bool*) – Include statements in export.
- **incl_markers** (*bool*) – Include markers in export.
- **incl_sql_tag** (*bool*) – Include the raw wrap in the sql that is rendered in the *md* export.

- **incl_exp_ctx** (*bool*) – Include (configurable) disclaimer at the top of exported *sql* file.

exported
List of file paths that current instance has exported to.

Type List[Path]

created
List of directory paths that current instance has created (should mostly apply for initial scaffolding build on first run only).

Type List[Path]

property export_dir(*self*) → pathlib.Path
Documentation sub-directory; *.snowmobile* by default.

property sections(*self*) → Dict[int, Section]
Dictionary of all *sections* by index position.

property markdown(*self*) → str
Full markdown file as a string.

property sql(*self*)
SQL for save.

save(*self*, *md: bool = True*, *sql: bool = True*) → None
Save files to disk.

Parameters

- **md** (*bool*) – Export a generated markdown file.
- **sql** (*bool*) – Export a generated sql file.

7.2.4 snowmobile.core.qa

Derived *Statement* classes.

These objects derive from *snowmobile.core.statement.Statement* and override its *process()* method to perform additional post-processing of the statement's results in conjunction with any parameters provided within the statement's tags.

`s.process()` modifies a statement's *outcome* attribute (`bool`) on which an assertion is run before continuing execution of the script.

Note: The `on_exception` and `on_failure` parameters of *script.run()* are passed directly and only applicable to these derived statement classes.

`on_exception` is used to control the exception-handling **of errors encountered in the post-processing invoked by** *s.process()*

`on_failure` is used to control the exception-handling **of a failed assertion ran on the outcome of the post-processing invoked by** *s.process()*

Module Contents

Classes

QA	Base class for QA st.
Empty	QA class for verification that a statement's results are empty.
Diff	QA class for comparison of values within a table based on

class snowmobile.core.qa.**QA**(*sn:* snowmobile.core.connection.Snowmobile, ***kwargs*)
Bases: *snowmobile.core.Statement*

Base class for QA st.

Initialize self. See help(type(self)) for accurate signature.

set_outcome(*self*)
Updates ._outcome upon completion of processing invoked by .process().

class snowmobile.core.qa.**Empty**(*sn:* snowmobile.core.connection.Snowmobile, ***kwargs*)
Bases: *snowmobile.core.qa.QA*

QA class for verification that a statement's results are empty.

The most widely applicable use of *Empty* is for simple verification that a table's dimensions are as expected.

Initialize self. See help(type(self)) for accurate signature.

process(*self*) → *snowmobile.core.qa.QA*
Over-ride method; checks if results are empty and updates outcome

class snowmobile.core.qa.**Diff**(*sn:* snowmobile.core.connection.Snowmobile = *None*, ***kwargs*)
Bases: *snowmobile.core.qa.QA*

QA class for comparison of values within a table based on partitioning on a field.

partition_on
Column name to partition data on before comparing the partitioned datasets; defaults to 'src_description`.

Type str

end_index_at
Column name that marks the last column to use as an index column when joining the partitioned datasets back together.

Type str

compare_patterns
Regex patterns to match columns on that should be *included* in comparison (numeric columns you're running QA on).

Type list

ignore_patterns
Regex patterns to match columns on that should be *ignored* both for the comparison and the index.

Type list

generic_metric_col_nm
Column name to use for the melted field names; defaults to 'Metric'.

Type str

compare_cols
Columns that are used in comparison once statement is executed and parsing is applied.

Type list

drop_cols
Columns that are dropped once statement is executed and parsing is applied.

Type list

idx_cols
Columns that are used for the index to join the data back together once statement is executed and parsing is applied.

Type list

ub_raw
Maximum absolute raw difference (upper bound) that two fields that are being compared can differ from each other without causing a failure.

Type float

ub_perc
Maximum absolute percentage difference (upper bound) that two comparison fields can differ from each other without causing a failure.

Type float

Instantiates a `qa-diff` statement.

Parameters

- **delta_column_suffix** (*str*) – Suffix to add to columns that comparison is being run on; defaults to 'Delta'.
- **partition_on** (*str*) – Column to partition the data on in order to compare.
- **end_index_at** (*str*) – Column name that marks the last column to use as an index when joining the partitioned datasets back together.
- **compare_patterns** (*list*) – Regex patterns matching columns to be *included* in comparison.
- **ignore_patterns** (*list*) – Regex patterns to match columns on that should be *ignored* both for the comparison and the index.
- **generic_metric_col_nm** (*str*) – Column name to use for the melted field names; defaults to 'Metric'.
- **raw_upper_bound** (*float*) – Maximum absolute raw difference that two fields that are being compared can differ from each other without causing a failure.
- **percentage_upper_bound** (*float*) – Maximum absolute percentage difference that two comparison fields can differ from each other without causing a failure.

split_cols(*self*) → *snowmobile.core.qa.Diff*
Post-processes results returned from a `qa-diff` statement.

Executes private methods to split columns into:

- Index columns
- Drop columns

- Comparison columns

Then runs checks needed to ensure minimum requirements are met in order for a valid partition/comparison to be made.

property partitioned_by (*self*) → Set[Any]
Distinct values within the `partition_on` column that data is partitioned by.

static partitions_are_equal (*partitions: Dict[str, pd.DataFrame]*, *abs_tol: float*, *rel_tol: float*) → bool
Evaluates if a dictionary of DataFrames are identical.

Parameters

- **partitions** (*Dict[str, pd.DataFrame]*) – A dictionary of DataFrames returned by `snowmobile.DataFrame()`.
- **abs_tol** (*float*) – Absolute tolerance for difference in any value amongst the DataFrames being compared.
- **rel_tol** (*float*) – Relative tolerance for difference in any value amongst the DataFrames being compared.

Returns (bool): Indication of equality amongst all the DataFrames contained in `partitions`.

process (*self*) → *snowmobile.core.qa.Diff*
Post-processing for *Diff*-specific results.

7.2.5 snowmobile.core.script

`snowmobile.Script` is instantiated from a local sql file or a readable text file containing valid sql code.

Module Contents

Classes

Script	Parser and operator of local sql files.

class snowmobile.core.script.**Script** (*sn: Optional[Snowmobile] = None*, *path: Optional[Path, str] = None*, *sql: Optional[str] = None*, *as_generic: bool = False*, *delay: bool = True*, ***kwargs*)
Bases: *snowmobile.core.Generic*

Parser and operator of local sql files.

Parameters

- **sn** (*snowmobile.core.connection.Snowmobile*) – An instance of *Snowmobile*.
- **path** (*Optional[str]*) – A full path to a sql file or readable text file containing valid sql code.
- **path** – A raw string of valid sql code as opposed to reading from a `path`.
- **as_generic** (*bool*) – Instantiate all statements as generic st; skips all checks for a mapping of a statement anchor to a derived statement class to instantiate in the place of a generic *Statement*.

- **delay** (*bool*) – Delay connection of the `Snowmobile`; only applicable if the `sn` argument is omitted and *`Script`* is instantiating a `Snowmobile` in its absence.
- ****kwargs** – Any keyword arguments to pass to `Snowmobile`; only applicable if the `sn` argument is omitted and *`Script`* is instantiating a `Snowmobile` in its absence

sn
An instance of *`Snowmobile`*

Type *snowmobile.core.connection.Snowmobile*

patterns
Configured patterns from *snowmobile.toml*.

Type *snowmobile.core.cfg.script.Pattern*

as_generic
Instantiate all statements as generic st; skips all checks for a mapping of a statement anchor to a derived statement class to instantiate in the place of a generic *`Statement`*.

Type bool

filters
Dictionary of filters that have been passed to the current instance of *`snowmobile.core.Script`*.

Type Dict[Any[str, int], Dict[str, Set]]

markers
Dictionary of all markers found in the script.

Type Dict[int, *cfg.Marker*]

path
Path to sql file (e.g. *full/path/to/script.sql*).

Type Path

name
Name of sql file (e.g. *script.sql*).

Type str

source
Raw sql text of script; will be the text contained in the raw sql file when initially read from source and reflect any modifications to the script's contents made post-instantiation.

Type str

read(*self*, *path: pathlib.Path = None*) → *snowmobile.core.script.Script*
Runs quick path validation and reads in a sql file as a string.

A valid *path* must be provided if the *script.path* attribute hasn't been set; `ValueErrors` will be thrown if neither is valid.

Parameters **path** (*`pathlib.Path`*) – Full path to a sql object.

from_str(*self*, *sql: str*, *name: str*, *directory: pathlib.Path = Path.cwd()*) → *snowmobile.core.script.Script*
Instantiates a raw string of sql as a script.

source(*self*, *original: bool = False*) → str
The script's sql as a raw string.

parse_one(*self*, *s: Union[sqlparse.sql.Statement, str]*, *index: Optional[int] = None*, *nm: Optional[str] = None*) → None
Adds a statement object to the script.

Default behavior will only add `sqlparse.sql.Statement` objects returned from `script.source_stream`.

`clean_parse()` utility function is utilized so that generated sql within Python can be inserted back into the script as raw strings.

Parameters

- **s** (*Union[sqlparse.sql.Statement, str]*) – A sqlparse.sql.Statement object or a raw string of SQL for an individual statement.
- **index** (*int*) – Index position of the statement within the script; defaults to `n + 1` if index is not provided where `n` is the number of statements within the script at the time `parse_one()` is called.
- **nm** (*Optional[str]*) – Optionally provided the name of the statement being added; the script instance will treat this value as if it were provided within an in-script wrap.

parse_stream(*self*, *stream: str*) → None

Parses a stream of sql and adds onto existing Script contents.

filter(*self*, *incl_kw: Optional[List[str], str] = None*, *incl_obj: Optional[List[str], str] = None*, *incl_desc: Optional[List[str], str] = None*, *incl_anchor: Optional[List[str], str] = None*, *incl_nm: Optional[List[str], str] = None*, *excl_kw: Optional[List[str], str] = None*, *excl_obj: Optional[List[str], str] = None*, *excl_desc: Optional[List[str], str] = None*, *excl_anchor: Optional[List[str], str] = None*, *excl_nm: Optional[List[str], str] = None*, *as_id: Optional[Union[str, int]] = None*, *from_id: Optional[Union[str, int]] = None*, *last: bool = False*) → ContextManager[*Script*]

Subset the script based on attributes of its st.

`script.filter()` returns a modified instance of script that can be operated on within the context defined.

Note: Keyword arguments beginning with `incl` or `excl` expect a string or a list of strings containing regex patterns with which to check for a match against the associated attribute of its st' `Name`.

Parameters

- **incl_kw** – Include only `kw`
- **incl_obj** – Include only `obj`
- **incl_desc** – Include only `desc`
- **incl_anchor** – Include only `anchor`
- **incl_nm** – Include only `nm`
- **excl_kw** – Exclude `kw`
- **excl_obj** – Exclude `obj`
- **excl_desc** – Exclude `desc`
- **excl_anchor** – Exclude `anchor`
- **excl_nm** – Exclude `nm`
- **as_id** – ID to assign the filters passed to method; used to populated the *filters* attribute

- **from_id** – ID previously used on the same instance of *Script* from which to populate filtered arguments
- **last** – Re-use the last set of filters passed to context manager.

Returns (Script): The instance of script based on the context imposed by arguments pr.

property depth (*self*) → int
Count of statements in the script.

property lines (*self*) → int
Number of lines in the script

property excluded (*self*)
All statements by index position excluded from the current context.

property executed (*self*) → Dict[int, Statement]
Executed statements by index position included in the current context.

reset (*self*, *index: bool = False*, *ctx_id: bool = False*, *in_context: bool = False*, *scope: bool = False*, *_filter: bool = False*) → *snowmobile.core.script.Script*
Resets indices and scope on all statements to their state as read from source.

Invoked before exiting *filter()* context manger to reverse the revised indices set by `index_to()` and inclusion/ exclusion scope set by `Statement.Name.scope()`.

property duplicates (*self*) → Dict[str, int]
Dictionary of indistinct statement names/tags within script.

s (*self*, *_id: Optional[str, int] = None*) → Any[Statement, Empty, Diff]
Fetch a single statement by _id.

property st (*self*) → Dict[Union[int, str], Statement]
Accessor for all statements.

dtl (*self*, *full: bool = False*, *excluded: bool = False*, *title: bool = True*, *r: bool = False*) → Union[str, None]
Prints summary of statements within the current scope to console.

property first_s (*self*)
First statement by index position.

property last_s (*self*)
Last statement by index position

property first (*self*) → Union[Statement, Empty, Diff]
First statement executed.

property last (*self*) → Union[Statement, Empty, Diff]
Last statement executed.

doc (*self*, *nm: Optional[str] = None*, *prefix: Optional[str] = None*, *suffix: Optional[str] = None*, *incl_markers: Optional[bool] = True*, *incl_sql: Optional[bool] = True*, *incl_exp_ctx: Optional[bool] = True*, *result_wrap: Optional[str] = None*) → *snowmobile.core.Markup*
Returns a *Markup* from the script.

Parameters

- **nm** (*Optional[str]*) – Alternate file name to use.
- **prefix** (*Optional[str]*) – Prefix for file name.
- **suffix** (*Optional[str]*) – Suffix for file name.

- **incl_markers** (*Optional[bool]*) – Include markers in exported files.
- **incl_sql** (*Optional[bool]*) – Include sql in exported files.
- **incl_exp_ctx** (*Optional[bool]*) – Include disclaimer of programmatic save in exported sql file.

Returns A *Markup* instance based on the contents included in the script's context.

ids(*self*, *_id: Optional[Union[Tuple, List]] = None*) → List[int]
Utility function to get a list of statement IDs given an *_id*.

Invoked within script.run() if the *_id* parameter is either a:

(1) tuple of integers (lower and upper bound of statement indices to run)

(2) list of integers or strings (statement names or indices to run)

(3) default=None; returns all statement indices within scope if so

Parameters **_id**(*Union[Tuple, List]*) – _id field provided to script.run() if it's neither an integer or a string.

Returns (List[int]): A list of statement indices to run.

run(*self*, *_id: Optional[str, int, Tuple[int, int], List] = None*, *as_df: bool = True*, *on_error: Optional[str] = None*, *on_exception: Optional[str] = None*, *on_failure: Optional[str] = None*, *lower: bool = True*, *render: bool = False*, ***kwargs*) → None
Performs statement-by-statement execution of the script's contents.

Executes script's contents that are included within its current context and any (optional) value passed to the `_id` argument.

Note: Keyword arguments `on_exception` and `on_failure` are only applicable to derived classes of *Statement* (e.g., those within *snowmobile.core.qa* by default).

Parameters

- **_id**(*Optional[str, int, Tuple[int, int], List]*) –

  Identifier for statement(s) to execute, can be either:

  - *None* (default); execute all statements
  - A single statement's *nm*
  - A single statement's index position
  - A tuple of lower/upper index bounds of statements to execute
  - A list of statement names or index positions to execute
- **as_df** (*bool*) – Store statement's results as a `DataFrame`; defaults to *True*
- **on_error** (*Optional[str]*) – Action to take on **execution** error; providing *c* will continue execution as opposed to raising exception.
- **on_exception** (*Optional[str]*) – Action to take on **post-processing** error from a derived *Statement*; providing *c* will continue execution as opposed to raising exception.

- **on_failure** (*Optional[str]*) – Action to take on **failure** of post-processing assertion from a derived *Statement*; providing *c* will continue execution as opposed to raising exception.
- **lower** (*bool*) – Lower-case columns in results returned if `as_df=True`.
- **render** (*bool*) – Render sql executed as markdown; only applicable in Jupyter/iPython environments.
- ****kwargs** –

items(*self*, *by_index: bool = True*, *ignore_scope: bool = False*, *statements: bool = True*, *markers: bool = False*, *validate: bool = True*) → ItemsView[Union[int, str], Union[Statement, Marker]]
Dunder items.

keys(*self*, ***kwargs*) → KeysView[Union[int, str]]
Access keys of items only.

values(*self*, ***kwargs*) → ValuesView[Union[int, str]]
Access values of items only.

dict(*self*, ***kwargs*) → Dict
Unpacking items view into an actual dictionary.

7.2.6 snowmobile.core.sql

SQL contains utility methods to generate common SQL commands; *Snowmobile* inherits everything from this object and passes along its *query()* method for statement execution.

Note: The *auto_run* attribute defaults to *True*, meaning that the generated sql will execute when a method is called; if set to *False* the method will return the sql as a string without executing.

The *SQL* object is primarily interacted with as a pre-instantiated attribute of `Snowmobile`; in these instances users can fetch the generated sql as a string either by:

1. Providing *run=False* to any method called; this will override all behavior set by the current value of `auto_run`
2. Setting the `auto_run` attribute to *False* on an existing instance of *SQL*, which will replicate the behavior of *(1)* without needing to provide *run=False* to each method called on that instance

---

Module Contents

Classes

SQL	SQL class for generation & execution of common sql commands.

class snowmobile.core.sql.**SQL**(*_query_func: Callable*, *_cfg:* snowmobile.core.configuration.Configuration, *nm: Optional[str] = None*, *schema: Optional[str] = None*, *obj: Optional[str] = None*, *auto_run: Optional[bool] = True*)
Bases: *snowmobile.core.Generic*

SQL class for generation & execution of common sql commands.

Intended to be interacted with as a parent of `Snowmobile`.

Note:

- All arguments except for `sn` are optional.
- The benefit of setting the other attributes on an instance of *SQL* is to (optionally) avoid passing the same information to multiple methods when generating a variety of statements around the same object.

---

nm
Object name to use in generated sql (e.g. 'some_table_name')

Type str

obj
Object type to use in generated sql (e.g. 'table')

Type str

schema
Schema to use when dot-prefixing sql; defaults to the schema with which the `sn` is connected to.

Type str

auto_run
Indicates whether to automatically execute the sql generated by a given method; defaults to *True*

Type bool

Initializes a `snowmobile.SQL` object.

info_schema(*self*, *loc: str*, *where: Optional[List[str]] = None*, *fields: Optional[List[str]] = None*, *order_by: Optional[List] = None*, *run: Optional[bool] = None*) → Union[str, pd.DataFrame]
Generic case of selecting from information schema location.

table_info(*self*, *nm: Optional[str] = None*, *fields: List[str] = None*, *restrictions: Dict[str, str] = None*, *order_by: List[Optional[str, int]] = None*, *all_schemas: bool = False*, *run: Optional[bool] = None*) → Union[str, pd.DataFrame]
Query `information_schema.tables` for a given table or view.

Parameters

- **nm** (*str*) – Table name, including schema if creating a stage outside of the current schema.
- **fields** (*List[str]*) – List of fields to include in returned results (e.g. ['table_name', 'table_type', 'last_altered'])
- **restrictions** (*List[str]*) – List of conditionals typed as literal components of a *where* clause (e.g. ["table_type = 'base table'", 'last_altered::date = current_date()']).
- **order_by** (*List[str]*) – List of fields or their ordinal positions to order the results by.
- **all_schemas** (*bool*) – Include tables/views from all schemas; defaults to *False*.
- **run** (*bool*) – Determines whether to run the generated sql or not; defaults to *None* which will reference the current value of the *auto_run* attribute which defaults to *True*.

Returns (Union[str, pd.DataFrame]):

Either:

1. The results of the query as a `pandas.DataFrame`, or
2. The generated query as a `str` of sql.

column_info(*self*, *nm: Optional[str] = None*, *fields: Optional[List] = None*, *restrictions: Optional[Dict] = None*, *order_by: Optional[List] = None*, *all_schemas: bool = False*, *run: Optional[bool] = None*) → Union[str, pd.DataFrame]

Query `information_schema.columns` for a given table or view.

Parameters

- **nm** (*str*) – Table name, including schema if creating a stage outside of the current schema.
- **fields** (*List[str]*) – List of fields to include in returned results (e.g. ['ordinal_position', 'column_name', 'data_type'])
- **restrictions** (*List[str]*) – List of conditionals typed as literal components of a *where* clause (e.g.["regexp_count(lower(column_name), 'tmstmp') = 0"]).
- **order_by** (*List[str]*) – List of fields or their ordinal positions to order the results by.
- **all_schemas** (*bool*) – Include tables/views from all schemas; defaults to *False*.
- **run** (*bool*) – Determines whether to run the generated sql or not; defaults to *None* which will reference the current value of the *auto_run* attribute which defaults to *True*.

Returns (Union[str, pd.DataFrame]):

Either:

1. The results of the query as a `pandas.DataFrame`, or
2. The generated query as a `str` of sql.

columns(*self*, *nm: Optional[str] = None*, *from_info_schema: bool = False*, *lower: bool = False*, *run: Optional[bool] = None*) → Union[str, List]

Returns an ordered list of columns for a table or view.

Note:

- Default behavior is to retrieve the columns for a table or view by selecting a single sample record and returning the column index from the DataFrame that's returned which is much faster than selecting the **column_names** from `information_schema.columns` pulling column names from the information schema
- This can be changed by passing *from_info_schema=True*.

Parameters

- **nm** (*str*) – Name of table or view, including schema if the table or view is outside of the current schema.
- **from_info_schema** (*bool*) – Indicates whether to retrieve columns via the `information_schema.columns` or by selecting a sample record from the table or view; defaults to *False*.

- **lower** (*bool*) – Lower case each column in the list that's returned.
- **run** (*bool*) – Execute generated sql; defaults to *True*, otherwise returns sql as a string.

Returns (Union[str, List]):

Either:

1. An ordered list of columns for the table or view, **or**
2. The query against the table or view as a `str` of sql.

select(*self*, *nm: Optional[str] = None*, *fields: Optional[List[str]] = None*, *apply: Optional[List[Tuple[str, str]]] = None*, *n: Optional[int] = None*, *run: Optional[bool] = None*, ***kwargs*) → Union[str, pd.DataFrame]

Generic *select* statement.

Parameters

- **nm** (*str*) – Table to select from, including schema if the table is outside of the current schema
- **fields** (*Optional[List[str]]*) – Select these fields (optional).
- **apply** (*Optional[List[Tuple[str, str]]]*) – Select aggregations of these fields.

  apply [(*this_func, to_this_field, [as_alias]*), (*.., .., [..]*),

  ]

  - `apply` should be provided as a list of tuples, each containing a minimum of 2 items (respectively) representing the aggregate function to apply and the field to which it should be applied
  - By default, the aggregated result inherits the name of the field being aggregated, including any qualifier (optionally) provided with the field name or an explicit alias included as a 3rd item within the tuple

    The following snippet exhaustively illustrates the functionality described above

```
sn.select(
   nm='sandbox.sample_table',
   apply=[
        ('count', 'col1'),
        ('count', 'distinct col1'),
        ('count', 'distinct col1', 'col1_dst'),
   ],
   run=False,
)
>>>
select
 count(col1)
   as col1
 ,count(distinct col1)
   as distinct_col1
 ,count(distinct col1)
   as col1_dst
from sandbox.sample_table
```

- **n** (*int*) – Number of records to return, implemented as a 'limit' clause in the query; defaults to 1.
- **run** (*bool*) – Execute generated sql; defaults to *True*, otherwise returns sql as a string.

Returns (Union[str, pd.DataFrame]): Either:

1. The results of the query as a `pandas.DataFrame`, or
2. The generated query as a `str` of sql.

exists (*self*, *nm: Optional[str] = None*) → bool
Checks the existence of a table or view.

Parameters **nm** (*str*) – Name of table or view, including schema if the table or view is outside of the current schema.

Returns (bool): Boolean indication of whether or not the table or view exists.

is_distinct (*self*, *nm: Optional[str] = None*, *field: Optional[str] = None*) → bool
Checks if table *nm* is distinct on column *on_col*

Parameters

- **nm** (*str*) – Table name.
- **field** (*str*) – Column name.

count (*self*, *nm: Optional[str] = None*, *of: Optional[str] = None*, *dst_of: Optional[str] = None*, *as_perc: Optional[bool] = None*, *run: Optional[bool] = None*) → Union[int, float]
Number of records within a table or view.

Parameters

- **nm** (*str*) – Table name, including schema if querying outside current schema.
- **of** (*str*) – Column name (indistinct).
- **dst_of** (*str*) – Column name (distinct).
- **as_perc** (*bool*) – Option to return distinct count of the *dst_of* column as a percentage of the namespace depth of the table or view.
- **run** (*bool*) – Execute generated sql; defaults to *True*, otherwise returns sql as a string.

Returns (Union[str, pd.DataFrame]):

Either:

1. The results of the query as a `pandas.DataFrame`, or
2. The generated query as a `str` of sql.

show (*self*, *obj: str*, *in_loc: Optional[str] = None*, *names: bool = False*, *run: Optional[bool] = None*, ***kwargs*) → Union[pd.DataFrame, List[str], str]
Show schema objects of typ 'obj', optionally 'in_loc'.

Parameters

- **obj** (*str*) – Schema object type ('tables', 'file formats', etc).
- **in_loc** (*str*) – Snowflake location ('in schema sandbox', 'in database prod', etc).

- **names** (*bool*) – Return a list of schema object names only ('name' field).
- **run** (*bool*) – Execute the generated sql or return it as a string.

Returns (Union[pd.DataFrame, str]):

Either:

1. The results of the query as a `pandas.DataFrame`
2. The 'names' column of the results returned as a list
3. The generated query as a `str` of sql

ddl (*self*, *nm: Optional[str] = None*, *obj: Optional[str] = None*, *run: Optional[bool] = None*) → str

Query the DDL for an schema object.

Parameters

- **nm** (*str*) – Name of the object to get DDL for, including schema if object is outside of the current schema.
- **obj** (*str*) – Type of object to get DDL for (e.g. 'table', 'view', 'file-format').
- **run** (*bool*) – Execute generated sql; defaults to *True*, otherwise returns sql as a string.

Returns (str):

Either:

1. The results of the query as a `pandas.DataFrame`, or
2. The generated query as a `str` of sql.

comment (*self*, *nm: Optional[str] = None*, *obj: Optional[str] = None*, *set_as: Optional[str] = None*, *from_json: bool = False*, *as_json: bool = False*, *run: Optional[bool] = None*, ***kwargs*) → Union[str, Dict]

Get or set comment on a schema object.

Parameters

- **nm** (*str*) – Name of the schema object, including schema prefix if object is outside implicit scope of the current connection.
- **obj** (*str*) – Type of schema object (e.g. 'table', 'schema', etc).
- **set_as** (*str*) – Content to set as comment on schema object.
- **from_json** (*bool*) – Parse schema object comment as a string of json and return it as a dictionary.
- **as_json** (*bool*) – Dump contents of 'set_as' to a string of json prior to setting comment.
- **run** (*bool*) – Execute generated sql; defaults to *True*, otherwise returns sql as a string.
- ****kwargs** – Keyword argument to pass to *json.loads(comment)* if *from_json=True*.

Returns (Union[str, pd.DataFrame]):

Either:

1. The schema object comment as a `str`

2. The generated query as a `str` of sql.
3. The schema object comment as a dictionary if *from_json=True*

last_altered(*self*, *nm: Optional[str] = None*, *run: Optional[bool] = None*) → Union[str, pd.Timestamp]

Last altered timestamp for a table or view.

Parameters

- **nm** (*str*) – Table name, including schema if creating a stage outside of the current schema.
- **run** (*bool*) – Execute generated sql; defaults to *True*, otherwise returns sql as a string.

Returns (Union[str, pd.DataFrame]):

Either:

1. The results of the query as a `pandas.DataFrame`, or
2. The generated query as a `str` of sql.

truncate(*self*, *nm: Optional[str] = None*, *run: Optional[bool] = None*) → Union[str, pd.DataFrame]

Truncate a table.

Parameters

- **nm** (*str*) – Name of table, including schema if the table is outside of the current schema.
- **run** (*bool*) – Execute generated sql; defaults to *True*, otherwise returns sql as a string.

Returns (Union[str, pd.DataFrame]):

Either:

1. The results of the query as a `pandas.DataFrame`, or
2. The generated query as a `str` of sql.

drop(*self*, *nm: Optional[str] = None*, *obj: Optional[str] = None*, *run: Optional[bool] = None*) → Union[str, pd.DataFrame]

Drop a `Snowflake` object.

Parameters

- **nm** (*str*) – Schema object's name.
- **obj** (*str*) – Type of schema object (e.g. 'table', 'view', or 'schema')
- **run** (*bool*) – Execute generated statement; defaults to *True*, otherwise returns sql as a string.

Returns (Union[str, pd.DataFrame]):

Either:

1. The results of the query as a `pandas.DataFrame`, or
2. The generated query as a `str` of sql.

clone(*self*, *nm: Optional[str] = None*, *to: Optional[str] = None*, *obj: Optional[str] = None*, *run: Optional[bool] = None*, *replace: bool = False*) → Union[str, pd.DataFrame]

Clone a `Snowflake` object.

Warning:

- Make sure to read Snowflake's documentation for restrictions and considerations when cloning objects.

Note:

- In this specific method, the value provided to `nm` and `to` can be a single object name, a single schema, or both in the form of *obj_schema.obj_name* depending on the desired outcome.
- Additionally, **at least one of the** `nm` **or** `to` **arguments must be pr**.
- The defaults for the target object are constructed such that users can **either**:
  1. Clone objects to *other* schemas that inherit the source object's *name* without specifying so in the `to` argument, **or**
  2. Clone objects within the *current* schema that inherit the source object's *schema* without specifying so in the `to` argument.
- If providing a schema without a name to either argument, prefix the value provided with __ to signify it's a schema and not a lower-level object to be cloned.
  - e.g. providing *nm='sample_table'* and *to='__sandbox'* will clone *sample_table* from the current schema to *sandbox.sample_table*.
- An assertion error will be raised raised if neither argument is specified as *this would result in a command to clone an object and store it in an object that has the same name & schema as the object being cloned*.

Parameters

- **nm** (*str*) – Name of the object to clone, including schema if cloning an object outside of the current schema.
- **to** (*str*) – Target name for cloned object, including schema if cloning an object outside of the current schema.
- **obj** (*str*) – Type of object to clone (e.g. 'table', 'view', 'file-format'); defaults to *table*.
- **run** (*bool*) – Execute generated sql; defaults to *True*, otherwise returns sql as a string.
- **replace** (*bool*) – Indicates whether to replace an existing stage if pre-existing; default is *False*.

Returns (Union[str, pd.DataFrame]):

Either:

1. The results of the query as a `pandas.DataFrame`, or
2. The generated query as a `str` of sql.

create_stage(*self*, *nm_stage: str*, *nm_format: str*, *replace: bool = False*, *run: Optional[bool] = None*) → Union[str, pd.DataFrame]

Create a staging table.

Parameters

- **nm_stage** (*str*) – Name of stage to create, including schema if creating a stage outside of the current schema.
- **nm_format** (*str*) – Name of file format to specify for the stage, including schema if using a format from outside of the current schema.
- **run** (*bool*) – Execute generated sql; defaults to *True*, otherwise returns sql as a string.
- **replace** (*bool*) – Indicates whether to replace an existing stage if pre-existing; default is *False*.

Returns (Union[str, pd.DataFrame]):

Either:

1. The results of the query as a `pandas.DataFrame`, or
2. The generated query as a `str` of sql.

put_file_from_stage(*self*, *path: Union[Path, str]*, *nm_stage: str*, *options: Optional[Dict] = None*, *ignore_defaults: bool = False*, *run: Optional[bool] = None*) → Union[str, pd.DataFrame]

Generates a 'put' command into a staging table from a local file.

Parameters

- **path** (*Union[Path, str]*) – Path to local data file as a `pathlib.Path` or string.
- **nm_stage** (*str*) – Name of the staging table to load into.
- **run** (*bool*) – Execute generated sql; defaults to *True*, otherwise returns sql as a string.
- **options** (*dict*) – Optional arguments to add to *put* statement in addition to the values specified in the `loading.put` section of **snowmobile.toml**.
- **ignore_defaults** (*bool*) – Option to ignore the values specified in **snowmobile.toml**; defaults to *False*.

Returns (Union[str, pd.DataFrame]):

Either:

1. The results of the query as a `pandas.DataFrame`, or
2. The generated query as a `str` of sql.

copy_into_table_from_stage(*self*, *nm: str*, *nm_stage: str*, *options: Optional[Dict] = None*, *ignore_defaults: bool = False*, *run: Optional[bool] = None*) → Union[str, pd.DataFrame]

Generates a command to copy data into a table from a staging table.

Parameters

- **nm** (*str*) – Name of the object to drop, including schema if creating a stage outside of the current schema.

- **nm_stage** (*str*) – Name of the staging table to load from.
- **run** (*bool*) – Execute generated sql; defaults to *True*, otherwise returns sql as a string.
- **options** (*dict*) – Optional arguments to add to *put* statement in addition to the values specified in the `loading.put` section of **snowmobile.toml**.
- **ignore_defaults** (*bool*) – Option to ignore the values specified in **snowmobile.toml**; defaults to *False*.

Returns (Union[str, pd.DataFrame]):

Either:

1. The results of the query as a `pandas.DataFrame`, or
2. The generated query as a `str` of sql.

current (*self*, *obj: str*, *run: Optional[bool] = None*) → Union[str, Union[str, int]]
Generic implementation of 'select current' for session-based objects.

Parameters

- **obj** (*str*) – Type of object to retrieve information for (schema, session, ..).
- **run** (*bool*) – Execute generated sql; defaults to *True*, otherwise returns sql as a string.

Returns (Union[str, pd.DataFrame]):

Either:

1. The results of the query as a `pandas.DataFrame`, or
2. The generated query as a `str` of sql.

current_session (*self*, *run: Optional[bool] = None*) → Union[str, pd.DataFrame]
Select the current session.

current_schema (*self*, *run: Optional[bool] = None*) → Union[str, pd.DataFrame]
Select the current schema.

current_database (*self*, *run: Optional[bool] = None*) → Union[str, pd.DataFrame]
Select the current database.

current_warehouse (*self*, *run: Optional[bool] = None*) → Union[str, pd.DataFrame]
Select the current warehouse.

current_role (*self*, *run: Optional[bool] = None*) → Union[str, pd.DataFrame]
Select the current role.

use (*self*, *obj: str*, *nm: str*, *run: Optional[bool] = None*)
Generic implementation of 'use' command for schema objects.

Parameters

- **nm** (*str*) – Name of object to use (schema name, warehouse name, role name, ..).
- **obj** (*str*) – Type of object to use (schema, warehouse, role, ..).
- **run** (*bool*) – Execute generated sql; defaults to *True*, otherwise returns sql as a string.

Returns (Union[str, pd.DataFrame]):

Either:

1. The results of the query as a `pandas.DataFrame`, or
2. The generated query as a `str` of sql.

use_schema (*self*, *nm: Optional[str] = None*, *run: Optional[bool] = None*) → Union[str, pd.DataFrame]
Use schema command.

use_database (*self*, *nm: Optional[str] = None*, *run: Optional[bool] = None*) → Union[str, pd.DataFrame]
Use database command.

use_warehouse (*self*, *nm: Optional[str] = None*, *run: Optional[bool] = None*) → Union[str, pd.DataFrame]
Use warehouse command.

use_role (*self*, *nm: Optional[str] = None*, *run: Optional[bool] = None*) → Union[str, pd.DataFrame]
Use role command.

static order (*by: List[Union[int, str]]*) → str
Generates 'order by' clause from a list of fields or field ordinal positions.

static where (*restrictions: Dict*) → str
Generates a 'where' clause based on a dictionary of restrictions.

Parameters **restrictions** (*dict*) – A dictionary of conditionals where each key/value pair respectively represents the left/right side of a condition within a 'where' clause.

Returns (str): Formatted where clause.

static fields (*fields: Optional[List[str]] = None*) → str
Utility to generate fields within a 'select' statement.

7.2.7 snowmobile.core.statement

Base class for all *Statement* objects.

Module Contents

Classes

Time	Container for execution time info.
Statement	Base class for all *Statement* objects.

class snowmobile.core.statement.**Time**(***data*)
Bases: pydantic.BaseModel

Container for execution time info.

started :int

ended :int

class snowmobile.core.statement.**Statement**(*sn:* snowmobile.core.connection.Snowmobile, *statement: Union[sqlparse.sql.Statement, str]*, *index: Optional[int] = None*, *attrs_raw: Optional[str] = None*, *e: Optional[ExceptionHandler] = None*, ***kwargs*)

Bases: *snowmobile.core.tag.Attrs*, *snowmobile.core.Name*, *snowmobile.core.Generic*

Base class for all *Statement* objects.

Home for attributes and methods that are associated with **all** statement objects, generic or QA.

sn
`snowmobile.connect` object.

Type snowmobile.connect

statement
A `sqlparse.sql.Statement` object.

Type Union[sqlparse.sql.Statement, str]

index
The context-specific index position of a statement within a script; can be *None*.

Type int

patterns
`config.Pattern` object for more succinct access to values specified in **snowmobile.toml**.

Type config.Pattern

results
The results of the statement if executed as a `pandas.DataFrame`.

Type pd.DataFrame

outcome
Numeric indicator of outcome; defaults to *0* and is modified based on the outcome of statement execution and/or QA validation for derived classes.

Type int

outcome_txt
Plain text of outcome ('skipped', 'failed', 'completed', 'passed').

Type str

outcome_html
HTML text for the outcome as an admonition/information banner based on the following mapping of *outcome_txt* to admonition argument:

- *failed* ——> *warning*
- *completed* –> *info*
- *passed* ——> *success*

Type str

start_time
Unix timestamp of the query start time if executed; 0 otherwise.

Type int

end_time
Unix timestamp of the query end time if executed; 0 otherwise.

Type int

execution_time
Execution time of the query in seconds if executed; 0 otherwise.

Type int

execution_time_txt
Plain text description of execution time if executed; returned in seconds if execution time is less than 60 seconds, minutes otherwise.

Type str

first_keyword
The first keyword within the statement as a `sqlparse.sql.Token`.

Type sqlparse.sql.Token

sql
The sql associated with the statement as a raw string.

Type str

Initialize self. See help(type(self)) for accurate signature.

sql(*self*, *set_as: Optional[str] = None*, *tag: bool = False*) → Union[str, *Statement*]
Raw sql from statement, including result limit if enabled.

parse(*self*) → Tuple[Dict, str]
Parses tag contents into a valid dictionary.

Uses the values specified in **snowmobile.toml** to parse a raw string of statement attributes into a valid dictionary.

Note:

- If `is_multiline` is *True* and *name* is not included within the arguments, an assertion error will be thrown.
- If `is_multiline` is *False*, the raw string within the wrap will be treated as the name.
- The `wrap` attribute is set once parsing is completed and name has been validated.

Returns (dict): Parsed wrap arguments as a dictionary.

start(*self*)
Sets *start_time* attribute.

end(*self*)
Updates execution time attributes.

In namespace, sets:

- *end_time*
- *execution_time*
- *execution_time_txt*

trim(*self*) → str
Statement as a string including only the sql and a single-line wrap name.

Note: The wrap name used here will be the user-pr wrap from the original script or a generated `Name.nm` if a wrap was not provided for a given statement.

---

property is_derived(*self*)
Indicates whether or not it's a generic or derived (QA) statement.

property lines(*self*) → List[str]
Returns each line within the statement as a list.

as_section(*self*, *incl_sql_tag: Optional[bool] = None*, *result_wrap: Optional[str] = None*) → *snowmobile.core.Section*
Returns current statement as a `Section` object.

set_state(*self*, *index: Optional[int] = None*, *ctx_id: Optional[int] = None*, *in_context: Optional[bool] = None*, *filters: dict = None*) → *snowmobile.core.statement.Statement*
Sets current state/context on a statement object.

Parameters

- **ctx_id** (*int*) – Unix timestamp the `script.filter()` context manager was invoked.
- **filters** (*dict*) – Kwargs passed to `script.filter()`.
- **index** (*int*) – Integer to set as the statement's index position.

reset(*self*, *index: bool = False*, *ctx_id: bool = False*, *in_context: bool = False*, *scope: bool = False*) → *snowmobile.core.statement.Statement*
Resets attributes on the statement object to reflect as if read from source.

In its current form, includes:

- Resetting the statement/wrap's index to their original values.
- Resetting the `is_included` attribute of the statement's `wrap` to *True*.
- Populating `error_last` with errors from current context.
- Caching current context's timestamp and resetting back to *None*.

process(*self*)
Used by derived classes for post-processing the returned results.

run(*self*, *as_df: bool = True*, *lower: bool = True*, *render: bool = False*, *on_error: Optional[str] = None*, *on_exception: Optional[str] = None*, *on_failure: Optional[str] = None*, *ctx_id: Optional[int] = None*) → *snowmobile.core.statement.Statement*
Run method for all statement objects.

Parameters

- **as_df** (*bool*) – Store results of query as `pandas.DataFrame` or `SnowflakeCursor`.
- **lower** (*bool*) – Lower case column names in *results* DataFrame if *as_df=True*.
- **render** (*bool*) – Render the sql executed as markdown.
- **on_error** (*str*) –

  Behavior if an execution/database error is encountered

  – *None*: default behavior, exception will be raised

  - *c*: continue with execution

- **on_exception** (*str*) – Behavior if an exception is raised in the **post-processing** of results from a derived class of *Statement* (`Empty` and `Diff`).
  - *None*: default behavior, exception will be raised
  - *c*: continue with execution
- **on_failure** (*str*) – Behavior if no error is encountered in execution or post-processing but the result of the post-processing has turned the statement's *outcome* attribute to False, indicating the results returned by the statement have failed validation.
  - *None*: default behavior, exception will be raised
  - *c*: continue with execution

Returns (Statement): Statement object post-executing query.

outcome_txt (*self*, *_id: Optional[int] = None*) → str
Outcome as a string.

property outcome_html (*self*) → str
Outcome as an html admonition banner.

7.2.8 snowmobile.core.table

`snowmobile.Table` is a canned implementation of the Bulk Loading from a Local File System standard and is intended to provide a predictable, no-nonsense method of loading a DataFrame, `df`, into a `table` (`str`).

Note: Core functionality includes:

1. Generating and executing generic DDL for `df` if the table doesn't yet exist
2. Executing DDL for the **file format** being used *if it doesn't yet exist in the current schema*, or (optionally) specifying an alias for a file format in its `file_format` argument; **in the case of the latter:**
   - An absolute `path` to an independent, user-defined sql file must be specified within the **external-sources.ddl** field of `snowmobile.toml`
   - Prior to attempting the load of `df`, `snowmobile.Table` will create a *Script* from the configured `path` and execute the (file format DDL) statement whose tagged name maps to the value provided to its `file_format` argument
   - An error will be thrown during the creation of the *Table* if the *Script* associated with the configured `path` does not contain a statement whose tagged name matches the value of `file_format` or if an error is raised when the file is parsed
   - Bypassed by creating the `Table` with:

```
snowmobile.Table(validate_format=False, **kwargs)
```

3. Dimensional compatibility checks between `df` and the table being loaded into
   - Bypassed by creating the `Table` with:

```
snowmobile.Table(validate_table=False, **kwargs)
```

4. Coercing column names of `df` into a generic database standard prior to loading, including de-duplication of field names when applicable
5. Argument or configuration based handling of action to take if table being loaded into already exists (respectively) via the `if_exists` argument to `snowmobile.Table()` or its associated section in `snowmobile.toml`; valid values are **replace**, **truncate**, **append**, **fail**

Module Contents

Classes

Table	Constructed with a `DataFrame` and a table name to load into.

class snowmobile.core.table.**Table**(*df: pandas.DataFrame*, *table: str*, *sn: Optional[Snowmobile] = None*, *if_exists: Optional[str] = None*, *as_is: bool = False*, *path_ddl: Optional[Union[str, Path]] = None*, *path_output: Optional[str, Path] = None*, *file_format: Optional[str] = None*, *incl_tmstmp: Optional[bool] = None*, *tmstmp_col_nm: Optional[str] = None*, *reformat_cols: Optional[bool] = None*, *validate_format: Optional[bool] = None*, *validate_table: Optional[bool] = None*, *upper_case_cols: Optional[bool] = None*, *lower_case_table: Optional[bool] = None*, *keep_local: Optional[bool] = None*, *on_error: Optional[str] = None*, *check_dupes: Optional[bool] = None*, *load_copy: Optional[bool] = None*, ***kwargs*)

Bases: *snowmobile.core.Generic*

Constructed with a `DataFrame` and a table name to load into.

The `df` and `table`'s compatibility can be inspected prior to calling the `Table.load()` method or by providing *as_is=True*` when instantiating the object; the latter will kick off the loading process invoked by *.load()* based on the parameters provided to *snowmobile.Table()*.

Parameters

- **df** (*DataFrame*) – The `DataFrame` to load.
- **table** (*str*) – The table name to load `df` into.
- **sn** (*Optional[Snowmobile]*) – An instance of `Snowmobile`; can be used to load a table on a specific connection or from a specific `snowmobile.toml` file.
- **if_exists** (*Optional[str]*) – Action to take if `table` already exists - options are *fail*, *replace*, *append*, and *truncate*; defaults to `append`.
- **as_is** (*bool*) – Load `df` into `table` based on the parameters provided to *Table* without further pre-inspection by the user; defaults to *False*.
- **path_ddl** (*Optional[Path]*) – Alternate path to file format DDL to use for load.
- **keep_local** (*Optional[bool]*) – Keep local file that is written out as part of the bulk loading process; defaults to *False*.
- **path_output** (*Optional[str Path]*) – Path to write output local file to; defaults to a generated file name exported in the current working directory.

- **file_format** (*Optional[str]*) – The name of the file_format to use when loading `df`; defaults to `snowmobile_default_psv`.
- **incl_tmstmp** (*Optional[bool]*) – Include timestamp of load as part of `table`; defaults to *True*.
- **tmstmp_col_nm** (*Optional[str]*) – Name to use for load timestamp if `incl_tmstmp=True`; defaults to *loaded_tmstmp*.
- **upper_case_cols** (*Optional[bool]*) – Upper case columns of `df` when loading into `table`; defaults to *True*.
- **reformat_cols** (*Optional[bool]*) – Reformat applicable columns of `df` to be DB-compliant; defaults to *True*.

  Reformatting primarily entails:

  - Replacing spaces and special characters with underscores
  - De-duping consecutive special characters
  - De-duping repeated column names; adds an `_i` suffix to duplicate fields where `i` is the nth duplicate name for a field

- **validate_format** (*Optional[bool]*) – Validate the file format being used prior to kicking off the load; defaults to *True*.

  Validation entails:

  - Checking if the file format being used already exists based on formats accessible to the current connection
  - Executing DDL for the file format being used if not, pulled from the `DDL` *ext-location* and the statement name `create file format~{format name}`

Tip: Providing *validate_format=False* will speed up loading time when batch-loading into an existing table by skipping this step

---

- **validate_table** (*Optional[bool]*) – Perform validations of `df` against `table` prior to kicking off the loading process; defaults to *True*.

  Validation entails:

  - Checking the existence of `table`; no further validation is performed if it does **not** exist
  - Compares the columns of `df` to the columns of `table` and stores results for use during loading process

---

Note: Table validation results are used in conjunction with the `if_exists` parameter to determine the desired behavior based on the (potential) existence of `table` and its compatibility with `df`.

Tip: Providing *validate_table=False* will speed up loading time time when batch-loading into an existing table

---

- **lower_case_table** (*Optional[bool]*) – Lower case `table` name; defaults to *False*.

- **on_error** (*Optional[str]*) – Action to take if an exception is encountered as part of the validating or loading process - providing `on_error='c'` will *continue* past an exception as opposed to raising it; defaults to *None* meaning any exception encountered will be raised
- **check_dupes** (*Optional[bool]*) – Check for duplicate field names in `df`; defaults to *True*.
- **load_copy** (*Optional[bool]*) – Alter and load a deep copy of `df` as opposed to the `df` in-memory as passed to the parameter; defaults to *True*.

db_responses
Responses from database during loading process.

Type Dict[str, str]

loaded
Table was loaded successfully.

Type bool

load(*self*, *if_exists: Optional[str] = None*, *from_script: pathlib.Path = None*, *verbose: bool = True*, ***kwargs*) → *snowmobile.core.table.Table*
Loads `df` into `table`.

Parameters

- **if_exists** (*Optional[str]*) – Determines behavior to take if the table being loaded into already exists; defaults to **append**; options are **replace**, **append**, **truncate**, and **fail**
- **from_script** (*Optional[Union[Path, str]]*) – Path to sql file containing custom DDL for `table`; DDL is assumed to have a valid statement name as is parsed by *Script* and following the naming convention of `create table~TABLE` where `TABLE` is equal to the value provided to the `table` keyword argument
- **verbose** (*bool*) – Verbose console output; defaults to **True**

Returns (Table): The *Table* after attempting load of `df` into `table`; a successful load can be verified by inspecting *loaded*

property exists(*self*) → bool
Indicates if the target table exists.

col_diff(*self*, *mismatched: bool = False*) → Dict[int, Tuple[str, str]]
Returns diff detail of local DataFrame to in-warehouse table.

property cols_match(*self*) → bool
Indicates if columns match between DataFrame and table.

load_statements(*self*, *from_script: pathlib.Path*) → List[str]
Generates exhaustive list of the statements to execute for a given instance of loading a DataFrame.

to_local(*self*, *quote_all: bool = True*) → None
Export to local file via configuration in `snowmobile.toml`.

property tm_load(*self*) → int
Seconds elapsed during loading.

property tm_validate_load(*self*) → int
Seconds elapsed during validation.

property tm_total(*self*) → int
Total seconds elapsed for load.

validate(*self*, *if_exists: str*) → None
Validates load based on current state through a variety of operations.

Parameters **if_exists** (*str*) – Desired behavior if table already exists; intended to be passed in from `table.load()` by default.

7.2.9 snowmobile.core.tag

Container for the attributes parsed from a `Tag`.

Module Contents

Classes

Attrs	Extended dictionary for attribute storage.

class snowmobile.core.tag.**Attrs**(*sn: Optional[Snowmobile] = None*, *raw: Optional[str] = None*, *args: Optional[str] = None*, *index: Optional[int] = None*, ***connection_kwargs*)
Bases: `dict`

Extended dictionary for attribute storage.

Initialize self. See help(type(self)) for accurate signature.

sn :Snowmobile
Connection / configuration.

Type Optional[snowmobile.Snowmobile]

index :int
Index position within the script.

Type int

tag(*self*, *raw: bool = False*, *namespace: bool = False*, *wrap: bool = False*) → Union[str, Dict, *Attrs*]
Explicit accessor for self.

property is_tagged(*self*) → bool
Statement has a prepended tag.

property is_multiline(*self*) → bool
Contains multiline wrap.

7.3 Package Contents

7.3.1 Classes

Generic	Generic dunder implementation for `snowmobile` objects.
ExceptionHandler	All snowmobile classes contain a *ExceptionHandler*.
Configuration	A parsed *snowmobile.toml* file.
Snowmobile	Primary method of statement execution and accessor to parsed snowmobile.toml.
connect	Primary method of statement execution and accessor to parsed snowmobile.toml.
Section	Represents any (1-6 level) header section within *Script Name (doc).md*.
Scope	Handles the scope/context for *Statement* objects and derived classes.
Name	Handles the decomposition/parsing of statement name.
Statement	Base class for all *Statement* objects.
Column	A single column within a `SnowFrame`.
Diff	QA class for comparison of values within a table based on
Empty	QA class for verification that a statement's results are empty.
SnowFrame	Extends a `DataFrame` with a `.snf` entry point.
SQL	SQL class for generation & execution of common sql commands.
Markup	Contains all sections within the context of a `Script`.
Script	Parser and operator of local sql files.
Table	Constructed with a `DataFrame` and a table name to load into.

class snowmobile.core.**Generic**

Bases: `object`

Generic dunder implementation for `snowmobile` objects.

Base class for all `snowmobile` objects that do **not** inherit from pydantic's BaseModel or configuration class, `Config`.

class snowmobile.core.**ExceptionHandler**(*within: snowmobile.core.errors.Optional[Any] = None*, *ctx_id: snowmobile.core.errors.Optional[int] = None*, *in_context: bool = False*, *children: snowmobile.core.errors.Dict[int, Any] = None*, *is_active_parent: bool = False*, *to_mirror: snowmobile.core.errors.Optional[List[Any]] = None*)

Bases: *snowmobile.core.Generic*

All snowmobile classes contain a *ExceptionHandler*.

Parameters

- **within** (*Optional[Any]*) – Class for which the ExceptionHandler is intended.

- **ctx_id** (*Optional[int]*) – Context ID; set/unset by methods when entering/exiting certain contexts.
- **in_context** (*bool*) – Class is currently within a specific *ctx_id*
- **children** (*Dict[int, Any]*) – Attributes of the `within` class for which the *ExceptionHandler* should mirror the methods called on the parent class. # TODO: Refactor this out; it's essentially janky multi inheritance
- **is_active_parent** (*bool*) – The `within` class is currently enforcing the context rules on its `children`
- **to_mirror** (*Optional[List[Any]]*) – Methods called in the *attr:`within* class that should be applied to its `children` (i.e. set/reset context ID, etc)

property current(*self*)
All exceptions in the current context.

collect(*self*, *e: Any[snowmobile_errors]*)
Stores an exception.

property first(*self*) → snowmobile.core.errors.Error
First exception encountered.

property last(*self*) → snowmobile.core.errors.Error
Last exception encountered.

seen(*self*, *from_ctx: snowmobile.core.errors.Optional[int] = None*, *of_type: snowmobile.core.errors.Optional[Any[snowmobile_errors], List[snowmobile_errors]] = None*, *to_raise: snowmobile.core.errors.Optional[bool] = None*, *with_ids: snowmobile.core.errors.Optional[int, List[int], Set[int]] = None*, *all_time: bool = False*) → bool
Boolean indicator of if an exception has been seen.

get(*self*, *from_ctx: snowmobile.core.errors.Optional[int] = None*, *of_type: snowmobile.core.errors.Optional[Any[snowmobile_errors], List[snowmobile_errors]] = None*, *to_raise: snowmobile.core.errors.Optional[bool] = None*, *with_ids: snowmobile.core.errors.Optional[int, List[int], Set[int]] = None*, *all_time: bool = False*, *last: bool = False*, *first: bool = False*, *_raise: bool = False*)
Boolean indicator of if an exception has been seen.

property ctx_id(*self*)
Current context id.

set(*self*, *ctx_id: snowmobile.core.errors.Optional[int] = None*, *in_context: bool = False*, *outcome: snowmobile.core.errors.Optional[int] = None*)
Set attributes on self.

set_from(*self*, *other: snowmobile.core.exception_handler.ExceptionHandler*) → snowmobile.core.exception_handler.ExceptionHandler
Updates attributes of self with those from 'other'.

reset(*self*, *ctx_id: bool = False*, *in_context: bool = False*, *outcome: bool = False*) → snowmobile.core.exception_handler.ExceptionHandler
Resets attributes on self.

property by_tmstmp(*self*)
All exceptions by timestamp, ordered by most to least recent.

class snowmobile.core.**Configuration**(*creds: Optional[str] = None*, *config_file_nm: Optional[str] = None*, *from_config: Optional[Path, str] = None*, *export_dir: Optional[Path, str] = None*, *silence: bool = False*)
Bases: `snowmobile.core.base.Generic`

A parsed *snowmobile.toml* file.

All keyword arguments optional.

Parameters

- **config_file_nm** (*Optional[str]*) – Name of configuration file to use; defaults to *snowmobile.toml*.
- **creds** (*Optional[str]*) – Alias for the set of credentials to authenticate with; default behavior will fall back to the *connection.default-creds* specified in *snowmobile.toml*, *or the first set of credentials stored if this configuration option is left blank*.
- **from_config** (*Optional[str, Path]*) – A full path to a specific configuration file to use; bypasses any checks for a cached file location and can be useful for container-based processes with restricted access to the local file system.
- **export_dir** (*Optional[Path]*) – Path to save a template *snowmobile.toml* file to; if pr, the file will be exported within the __init__ method and nothing else will be instantiated.

file_nm
Configuration file name; defaults to 'snowmobile.toml'.

Type str

cache
Persistent cache; caches *location*.

Type snowmobile.core.cache.Cache

location
Full path to configuration file.

Type pathlib.Path

connection :Optional[cfg.Connection]
[connection] from snowmobile.toml.

Type *snowmobile.core.cfg.Connection*

loading :Optional[cfg.Loading]
[loading] from snowmobile.toml.

Type *snowmobile.core.cfg.Loading*

script :Optional[cfg.Script]
[script] from snowmobile.toml.

Type *snowmobile.core.cfg.Script*

sql :Optional[cfg.SQL]
[sql] from snowmobile-ext.toml.

Type *snowmobile.core.cfg.SQL*

ext_sources :Optional[cfg.Location]
[external-sources] from snowmobile.toml.

Type *snowmobile.core.cfg.Location*

property markdown(*self*) → *snowmobile.core.cfg.Markup*
Accessor for cfg.script.markdown.

property attrs(*self*) → *snowmobile.core.cfg.Attributes*
Accessor for cfg.script.markdown.attributes.

property wildcards (*self*) → *snowmobile.core.cfg.Wildcard*
Accessor for cfg.script.patterns.wildcards.

static batch_set_attrs (*obj: Any*, *attrs: dict*, *to_none: bool = False*)
Batch sets attributes on an object from a dictionary.

Parameters

- **obj** (*Any*) – Object to set attributes on.
- **attrs** (*dict*) – Dictionary containing attributes.
- **to_none** (*bool*) – Set all of the object's attributes batching a key in *wrap* to *None*; defaults ot *False*.

Returns (Any): Object post-setting attributes.

static attrs_from_obj (*obj: Any*, *within: Optional[List[str]] = None*) → Dict[str, MethodType]
Utility to return attributes/properties from an object as a dictionary.

static methods_from_obj (*obj: Any*, *within: Optional[List[str]] = None*) → Dict[str, MethodType]
Returns callable components of an object as a dictionary.

property scopes (*self*)
All combinations of scope type and scope attribute.

scopes_from_kwargs (*self*, *only_populated: bool = False*, ***kwargs*) → Dict
Turns *script.filter()* arguments into a valid set of kwargs for *Scope*.

Returns dictionary of all combinations of 'arg' ("kw", "obj", "desc", "anchor" and "nm"), including empty sets for any 'arg' not included in the keyword arguments provided.

scopes_from_tag (*self*, *t: Any*)
Generates list of keyword arguments to instantiate all scopes for a wrap.

json (*self*, *by_alias: bool = False*, ***kwargs*)
Serialization method for core object model.

class snowmobile.core.**Snowmobile** (*creds: Optional[str] = None*, *delay: bool = False*, *ensure_alive: bool = True*, *config_file_nm: Optional[str] = None*, *from_config: Optional[str, Path] = None*, *silence: bool = False*, ***connect_kwargs*)
Bases: *snowmobile.core.sql.SQL*

Primary method of statement execution and accessor to parsed snowmobile.toml.

Parameters

- **creds** (*Optional[str]*) – Alias for the set of credentials to authenticate with; default behavior will fall back to the `connection.default-creds` specified in *snowmobile.toml*, *or the first set of credentials stored if this configuration option is left blank*.
- **delay** (*bool*) – Optionally delay establishing a connection when the object is instantiated, enabling access to the configuration object model through the `Connection.cfg` attribute; defaults to *False*.
- **ensure_alive** (*bool*) – Establish a new connection if a method requiring a connection against the database is called while *alive* is *False*; defaults to *True*.
- **config_file_nm** (*Optional[str]*) – Name of configuration file to use; defaults to *snowmobile.toml*.

- **from_config** (*Optional[str, Path]*) – A full path to a specific configuration file to use; bypasses any checks for a cached file location and can be useful for container-based processes with restricted access to the local file system.
- ****connect_kwargs** – Additional arguments to provide to snowflake.connector.connect(); arguments provided here will over-ride connection arguments specified in *snowmobile.toml*, including:
  - Connection parameters in *connection.default-arguments*
  - Credentials parameters associated with a given alias
  - Connection parameters associated with a given alias

Initializes a `snowmobile.SQL` object.

cfg :Configuration
snowmobile.toml

Type *snowmobile.core.configuration.Configuration*

con :Optional[SnowflakeConnection]
Can be *None* until set by *Snowmobile.connect()*

Type SnowflakeConnection

e :ExceptionHandler
Exception / context management

Type snowmobile.core.exception_handler.ExceptionHandler

ensure_alive :bool
Reconnect to Snowflake if connection is lost

Type bool

connect(*self*, ***kwargs*) → *snowmobile.core.connection.Snowmobile*
Establishes connection to Snowflake.

Re-implements *snowflake.connector.connect()* with connection arguments sourced from snowmobile's object model, specifically:

- Credentials from *snowmobile.toml*.
- Default connection arguments from *snowmobile.toml*.
- Optional keyword arguments either passed to `snowmobile.connect()` or directly to this method.

kwargs: Optional keyword arguments to pass to snowflake.connector.connect(); arguments passed here will over-ride `connection.default-arguments` specified in `snowmobile.toml`.

disconnect(*self*) → *snowmobile.core.connection.Snowmobile*
Disconnect from connection with which Connection() was instantiated.

property alive(*self*) → bool
Check if the connection is alive.

property cursor(*self*) → snowflake.connector.connection.SnowflakeCursor
`SnowflakeCursor` accessor.

property dictcursor(*self*) → snowflake.connector.DictCursor
`DictCursor` accessor.

ex(*self*, *sql: str*, *on_error: Optional[str] = None*, ***kwargs*) → snowflake.connector.connection.SnowflakeCursor
Executes a command via `SnowflakeCursor`.

Parameters

- **sql** (*str*) – `sql` command as a string.
- **on_error** (*str*) – String value to impose a specific behavior if an error occurs during the execution of `sql`.
- ****kwargs** – Optional keyword arguments for `SnowflakeCursor.execute()`.

Returns (SnowflakeCursor): `SnowflakeCursor` object that executed the command.

exd(*self*, *sql: str*, *on_error: Optional[str] = None*, ***kwargs*) → snowflake.connector.DictCursor
Executes a command via `DictCursor`.

Parameters

- **sql** (*str*) – `sql` command as a string.
- **on_error** (*str*) – String value to impose a specific behavior if an error occurs during the execution of `sql`.
- ****kwargs** – Optional keyword arguments for `SnowflakeCursor.execute()`.

Returns (DictCursor): `DictCursor` object that executed the command.

query(*self*, *sql: str*, *as_df: bool = False*, *as_cur: bool = False*, *as_dcur: bool = False*, *as_scalar: bool = False*, *lower: bool = True*, *on_error: Optional[str] = None*) → Union[pd.DataFrame, SnowflakeCursor]
Execute a query and return results.

Default behavior of *results=True* will return results as a `pandas.DataFrame`, otherwise will execute the sql provided with a `SnowflakeCursor` and return the cursor object.

Parameters

- **sql** (*str*) – Raw SQL to execute.
- **as_df** (*bool*) – Return results in DataFrame.
- **as_cur** (*bool*) – Return results in Cursor.
- **as_dcur** (*bool*) – Return results in a DictCursor.
- **as_scalar** (*bool*) – Return results as a single scalar value.
- **lower** (*bool*) – Boolean value indicating whether or not to return results with columns lower-cased.
- **on_error** (*str*) – String value to impose a specific behavior if an error occurs during the execution of `sql`.

Returns (Union[pd.DataFrame, SnowflakeCursor]): Results from `sql` as a `DataFrame` by default or the `SnowflakeCursor` object if *results=False*.

class snowmobile.core.**connect**(*creds: Optional[str] = None*, *delay: bool = False*, *ensure_alive: bool = True*, *config_file_nm: Optional[str] = None*, *from_config: Optional[str, Path] = None*, *silence: bool = False*, ***connect_kwargs*)
Bases: *snowmobile.core.sql.SQL*

Primary method of statement execution and accessor to parsed snowmobile.toml.

Parameters

- **creds** (*Optional[str]*) – Alias for the set of credentials to authenticate with; default behavior will fall back to the `connection.default-creds` specified in *snowmobile.toml*, *or the first set of credentials stored if this configuration option is left blank*.
- **delay** (*bool*) – Optionally delay establishing a connection when the object is instantiated, enabling access to the configuration object model through the `Connection.cfg` attribute; defaults to *False*.
- **ensure_alive** (*bool*) – Establish a new connection if a method requiring a connection against the database is called while *alive* is *False*; defaults to *True*.
- **config_file_nm** (*Optional[str]*) – Name of configuration file to use; defaults to *snowmobile.toml*.
- **from_config** (*Optional[str, Path]*) – A full path to a specific configuration file to use; bypasses any checks for a cached file location and can be useful for container-based processes with restricted access to the local file system.
- ****connect_kwargs** – Additional arguments to provide to snowflake.connector.connect(); arguments provided here will over-ride connection arguments specified in *snowmobile.toml*, including:
  - Connection parameters in *connection.default-arguments*
  - Credentials parameters associated with a given alias
  - Connection parameters associated with a given alias

Initializes a `snowmobile.SQL` object.

cfg :Configuration
snowmobile.toml

Type *snowmobile.core.configuration.Configuration*

con :Optional[SnowflakeConnection]
Can be *None* until set by *Snowmobile.connect()*

Type SnowflakeConnection

e :ExceptionHandler
Exception / context management

Type snowmobile.core.exception_handler.ExceptionHandler

ensure_alive :bool
Reconnect to Snowflake if connection is lost

Type bool

connect(*self*, ***kwargs*) → *snowmobile.core.connection.Snowmobile*
Establishes connection to Snowflake.

Re-implements *snowflake.connector.connect()* with connection arguments sourced from snowmobile's object model, specifically:

- Credentials from *snowmobile.toml*.
- Default connection arguments from *snowmobile.toml*.
- Optional keyword arguments either passed to `snowmobile.connect()` or directly to this method.

kwargs: Optional keyword arguments to pass to snowflake.connector.connect(); arguments passed here will over-ride `connection.default-arguments` specified in `snowmobile.toml`.

disconnect(*self*) → *snowmobile.core.connection.Snowmobile*
Disconnect from connection with which Connection() was instantiated.

property alive(*self*) → bool
Check if the connection is alive.

property cursor(*self*) → snowflake.connector.connection.SnowflakeCursor
`SnowflakeCursor` accessor.

property dictcursor(*self*) → snowflake.connector.DictCursor
`DictCursor` accessor.

ex(*self*, *sql: str*, *on_error: Optional[str] = None*, ***kwargs*) → snowflake.connector.connection.SnowflakeCursor
Executes a command via `SnowflakeCursor`.

Parameters

- **sql** (*str*) – `sql` command as a string.
- **on_error** (*str*) – String value to impose a specific behavior if an error occurs during the execution of `sql`.
- ****kwargs** – Optional keyword arguments for `SnowflakeCursor.execute()`.

Returns (SnowflakeCursor): `SnowflakeCursor` object that executed the command.

exd(*self*, *sql: str*, *on_error: Optional[str] = None*, ***kwargs*) → snowflake.connector.DictCursor
Executes a command via `DictCursor`.

Parameters

- **sql** (*str*) – `sql` command as a string.
- **on_error** (*str*) – String value to impose a specific behavior if an error occurs during the execution of `sql`.
- ****kwargs** – Optional keyword arguments for `SnowflakeCursor.execute()`.

Returns (DictCursor): `DictCursor` object that executed the command.

query(*self*, *sql: str*, *as_df: bool = False*, *as_cur: bool = False*, *as_dcur: bool = False*, *as_scalar: bool = False*, *lower: bool = True*, *on_error: Optional[str] = None*) → Union[pd.DataFrame, SnowflakeCursor]
Execute a query and return results.

Default behavior of *results=True* will return results as a `pandas.DataFrame`, otherwise will execute the sql provided with a `SnowflakeCursor` and return the cursor object.

Parameters

- **sql** (*str*) – Raw SQL to execute.
- **as_df** (*bool*) – Return results in DataFrame.
- **as_cur** (*bool*) – Return results in Cursor.
- **as_dcur** (*bool*) – Return results in a DictCursor.

- **as_scalar** (*bool*) – Return results as a single scalar value.
- **lower** (*bool*) – Boolean value indicating whether or not to return results with columns lower-cased.
- **on_error** (*str*) – String value to impose a specific behavior if an error occurs during the execution of `sql`.

Returns (Union[pd.DataFrame, SnowflakeCursor]): Results from `sql` as a `DataFrame` by default or the `SnowflakeCursor` object if *results=False*.

class snowmobile.core.**Section**(*cfg: snowmobile.core.Configuration*, *is_marker: bool = None*, *h_contents: Optional[str] = None*, *index: Optional[int] = None*, *parsed: Optional[Dict] = None*, *raw: Optional[str] = None*, *sql: Optional[str] = None*, *results: Optional[pd.DataFrame] = None*, *incl_sql_tag: bool = False*, *is_multiline: bool = False*, *result_wrap: Optional[str] = None*)

Bases: *snowmobile.core.Generic*

Represents any (1-6 level) header section within *Script Name (doc).md*.

Class is created with a call to the *as_section()* method or by the *snowmobile.core.markup.Markup* class in the case of a *Marker*.

In order to include execution metadata if available without sacrificing base-case parsing, the below implementation heavily relies properties over attributes to reconcile what's populated in the `st` vs `executed` attributes of *Script*.

Parameters

- **is_marker** (*bool*) – Information provided is associated with a marker as opposed to a statement; defaults to *False*.
- **h_contents** (*str*) – String representation of header contents.
- **index** (*int*) – Statement index position or *None* if marker.
- **parsed** (*dict*) – Parsed arguments from the statement or marker within the script.
- **raw** (*str*) – Raw wrap as `parsed` was parsed from.
- **sql** (*str*) – Statement's raw sql or *None* if marker.
- **results** (*pd.DataFrame*) – Results returned by statement's sql as a `DataFrame`; will be *None* if statement hasn't been executed or if a marker.

hx

String representation of header level (e.g. '#' for h1), based on the script/statement header-level specifications in *snowmobile.toml*.

Type str

Instantiation of a `script.Section` object.

reorder_attrs(*self*, *parsed: dict*, *cfg: snowmobile.core.Configuration*) → Dict

Re-orders parsed attributes based on configuration.

parse_contents(*self*, *cfg: snowmobile.core.Configuration*) → List[Item]

Unpacks sorted dictionary of parsed attributes into formatted Items.

property header(*self*) → str

Constructs the header for a section.

Uses specifications in *snowmobile.toml* to determine:

(1) The level of the header depending on whether it's a statement section or a script section.

(2) Whether or not to include the statement index as part of the header.

Returns Formatted header line as a string.

property sql_md(*self*) → str
Returns renderable sql or an empty string if script-level section.

property body(*self*) → str
All section content except for header.

property md(*self*) → str
Constructs a full section as a string from various components.

Returns Full string of valid markdown for the section.

class snowmobile.core.**Scope**(*arg: str*, *base: str*)
Bases: *snowmobile.core.Generic*

Handles the scope/context for *Statement* objects and derived classes.

Should never be interacted with from the user-facing API.

base
The left-most word within a statement wrap. For **generic** st this will be the *keyword* and for **QA** statements this will be the literal word `qa`.

Type str

component
The component within a given wrap that is being evaluated; this will be exactly **one** of *kw*, *obj*, *anchor*, *desc*, or *nm*.

Type str

incl_arg
The keyword argument that would be used to exclude a given component;

- e.g. if *component* is *kw*, *incl_arg* would be `incl_kw`.

Type str

excl_arg
The keyword argument that would be used to exclude a given component; this would be the same as the above example except the value would be `excl_kw` as opposed to `incl_kw`.

Type str

fallback_to

The default values to fall back to for *incl_arg* and *excl_arg* if they are not passed as a keyword argument by the user in *Script*; defaults to including the `base` and excluding an empty list.

type dict

provided_args (dict):

The set of keyword arguments provided at the time of the last call to *eval()*.

check_against_args (dict): The set of keyword arguments checked against at the time of the last call to *eval()*; will use provided arguments if they exist and the arguments from *fallback_to* otherwise.

is_included (bool): Name is included based on the results of the last call to *eval()*.

is_excluded (bool): Name is excluded based on the results of the last call to *eval()*.

Instantiates a *Scope* object.

parse_kwargs(*self*, ***kwargs*) → None

Parses all filter arguments looking for those that match its base.

Looks for include/exclude arguments within kwargs, populating `provided_args` with those that were provided and populates `check_against_args` with the same values if they were provided and fills in defaults from *fallback_to* otherwise.

Parameters ****kwargs** – Keyword arguments passed to *Script.filter()* (e.g. *incl_kw*, *excl_kw*, ..)

matches_patterns(*self*, *arg: str*) → bool

Returns indication of if `base` matches a given set of patterns.

Parameters **arg** (*str*) – Will either be the value of *incl_arg* or `exclude_arg`.

Returns (bool): Indication of whether any matches were found.

property included(*self*)

Name is included based on results of last *eval()*.

eval(*self*, ***kwargs*) → bool

Evaluates filter arguments and updates context accordingly.

Updates the values of `is_included`, `is_excluded`, and *included*.

Parameters ****kwargs** – Keyword arguments passed to *Script.filter()* (e.g. *incl_kw*, *excl_kw*, ..)

Returns (bool): Indicator of whether or not the statement should be included/excluded based on the context/keyword arguments pr.

class snowmobile.core.**Name**(*configuration: snowmobile.core.Configuration*, *nm_pr: Optional[str] = None*, *sql: Optional[str] = None*, *index: Optional[int] = None*)

Bases: *snowmobile.core.Generic*

Handles the decomposition/parsing of statement name.

Should never be instantiated directly by the user-facing API but its attributes are likely to be accessed often as part of *Statement* and derived classes.

cfg

`snowmobile.Configuration` object; represents fully parsed **snowmobile.toml** file.

Type snowmobile.Configuration

patt

`snowmobile.Schema.Pattern` object; represents `script.patterns` section of **snowmobile.toml**.

Type snowmobile.Schema.Pattern

_nm_pr

Provided wrap name for a given *Statement*; can be empty.

Type str

index
Statement index position within *Script*; can be empty.

Type int

is_included
Indicator of whether or not the combination of all scopes for this statement wrap is included within a given context.

Type bool

incl_idx_in_desc
Indicator of whether or not to include the statement index in the *description* component of the wrap; defaults to *True* so that all generated statement tags are guaranteed to be unique for a given script.

- Mainly included for testing purposes where setting to *False* enables comparing generated to provided statement tags without having to change the index position of the hard-coded/pr statement wrap when adding/removing tests.

Type bool

first_line_remainder
The remainder of the first line once excluding the `first_keyword` and stripping repeating whitespace.

Type str

scopes
Combination of all scopes for a given wrap; this is essentially the all possible combinations of including/excluding any of the *kw*, *nm*, *obj*, *desc*, and *anchor* for a given instance of *Name*.

Type set[*Scope*]

scope(*self*, ***kwargs*) → bool
Evaluates all component's of a wrap's scope against a set of filter args.

****kwargs:** Keyword arguments passed to *Script.filter()* (e.g. *incl_kw*, *excl_kw*, ..)

Returns (bool): Value indicating whether or not the statement should be included based on the outcome of the evaluation of all of its components.

nm(*self*, *ge: bool = False*, *pr: bool = False*, *og: bool = True*) → str
The final statement's **name** that is used by the API.

This will be the full statement name if a tag exists and a parsed/generated name otherwise.

kw(*self*, *ge: bool = False*, *pr: bool = False*)
The final statement's **keyword** that is used by the API.

This will be the provided keyword if a statement wrap exists and a parsed/ge keyword otherwise.

obj(*self*, *ge: bool = False*, *pr: bool = False*)
The final statement's **object** that is used by the API.

This will be the object within a wrap if a statement wrap exists and follows the correct structure and a parsed/ge object otherwise.

desc(*self*, *ge: bool = False*, *pr: bool = False*)
The final statement's **description** that is used by the API.

This will be the description within a wrap if a statement wrap exists and follows the correct structure and a parsed/ge description otherwise.

anchor(*self*, *ge: bool = False*, *pr: bool = False*)
The final statement's **anchor** that is used by the API.

This will be the anchor within a wrap if a statement wrap exists and follows the correct structure and a parsed/ge wrap name otherwise.

set(*self*, *key*, *value*) → snowmobile.core.name.Name
Custom attribute setting.

class snowmobile.core.**Statement**(*sn: snowmobile.core.connection.Snowmobile*, *statement: Union[sqlparse.sql.Statement, str]*, *index: Optional[int] = None*, *attrs_raw: Optional[str] = None*, *e: Optional[ExceptionHandler] = None*, ***kwargs*)
Bases: *snowmobile.core.tag.Attrs*, *snowmobile.core.Name*, *snowmobile.core.Generic*

Base class for all *Statement* objects.

Home for attributes and methods that are associated with **all** statement objects, generic or QA.

sn
`snowmobile.connect` object.

Type snowmobile.connect

statement
A `sqlparse.sql.Statement` object.

Type Union[sqlparse.sql.Statement, str]

index
The context-specific index position of a statement within a script; can be *None*.

Type int

patterns
`config.Pattern` object for more succinct access to values specified in **snowmobile.toml**.

Type config.Pattern

results
The results of the statement if executed as a `pandas.DataFrame`.

Type pd.DataFrame

outcome
Numeric indicator of outcome; defaults to *0* and is modified based on the outcome of statement execution and/or QA validation for derived classes.

Type int

outcome_txt
Plain text of outcome ('skipped', 'failed', 'completed', 'passed').

Type str

outcome_html
HTML text for the outcome as an admonition/information banner based on the following mapping of *outcome_txt* to admonition argument:

- *failed* ——> *warning*
- *completed* –> *info*
- *passed* ——> *success*

Type str

start_time
Unix timestamp of the query start time if executed; 0 otherwise.

Type int

end_time
Unix timestamp of the query end time if executed; 0 otherwise.

Type int

execution_time
Execution time of the query in seconds if executed; 0 otherwise.

Type int

execution_time_txt
Plain text description of execution time if executed; returned in seconds if execution time is less than 60 seconds, minutes otherwise.

Type str

first_keyword
The first keyword within the statement as a `sqlparse.sql.Token`.

Type sqlparse.sql.Token

sql
The sql associated with the statement as a raw string.

Type str

Initialize self. See help(type(self)) for accurate signature.

sql(*self*, *set_as: Optional[str] = None*, *tag: bool = False*) → Union[str, *Statement*]
Raw sql from statement, including result limit if enabled.

parse(*self*) → Tuple[Dict, str]
Parses tag contents into a valid dictionary.

Uses the values specified in **snowmobile.toml** to parse a raw string of statement attributes into a valid dictionary.

Note:

- If `is_multiline` is *True* and *name* is not included within the arguments, an assertion error will be thrown.
- If `is_multiline` is *False*, the raw string within the wrap will be treated as the name.
- The `wrap` attribute is set once parsing is completed and name has been validated.

Returns (dict): Parsed wrap arguments as a dictionary.

start(*self*)
Sets *start_time* attribute.

end(*self*)
Updates execution time attributes.

In namespace, sets:

- *end_time*
- *execution_time*
- *execution_time_txt*

trim(*self*) → str
Statement as a string including only the sql and a single-line wrap name.

Note: The wrap name used here will be the user-pr wrap from the original script or a generated *Name.nm* if a wrap was not provided for a given statement.

---

property is_derived(*self*)
Indicates whether or not it's a generic or derived (QA) statement.

property lines(*self*) → List[str]
Returns each line within the statement as a list.

as_section(*self*, *incl_sql_tag: Optional[bool] = None*, *result_wrap: Optional[str] = None*) → *snowmobile.core.Section*
Returns current statement as a *Section* object.

set_state(*self*, *index: Optional[int] = None*, *ctx_id: Optional[int] = None*, *in_context: Optional[bool] = None*, *filters: dict = None*) → *snowmobile.core.statement.Statement*
Sets current state/context on a statement object.

Parameters

- **ctx_id** (*int*) – Unix timestamp the `script.filter()` context manager was invoked.
- **filters** (*dict*) – Kwargs passed to `script.filter()`.
- **index** (*int*) – Integer to set as the statement's index position.

reset(*self*, *index: bool = False*, *ctx_id: bool = False*, *in_context: bool = False*, *scope: bool = False*) → *snowmobile.core.statement.Statement*
Resets attributes on the statement object to reflect as if read from source.

In its current form, includes:

- Resetting the statement/wrap's index to their original values.
- Resetting the `is_included` attribute of the statement's `wrap` to *True*.
- Populating `error_last` with errors from current context.
- Caching current context's timestamp and resetting back to *None*.

process(*self*)
Used by derived classes for post-processing the returned results.

run(*self*, *as_df: bool = True*, *lower: bool = True*, *render: bool = False*, *on_error: Optional[str] = None*, *on_exception: Optional[str] = None*, *on_failure: Optional[str] = None*, *ctx_id: Optional[int] = None*) → *snowmobile.core.statement.Statement*
Run method for all statement objects.

Parameters

- **as_df** (*bool*) – Store results of query as `pandas.DataFrame` or `SnowflakeCursor`.
- **lower** (*bool*) – Lower case column names in *results* DataFrame if *as_df=True*.
- **render** (*bool*) – Render the sql executed as markdown.

- **on_error** (*str*) –

  Behavior if an execution/database error is encountered

  - *None*: default behavior, exception will be raised
  - *c*: continue with execution

- **on_exception** (*str*) – Behavior if an exception is raised in the **post-processing** of results from a derived class of *Statement* (*Empty* and *Diff*).
  - *None*: default behavior, exception will be raised
  - *c*: continue with execution

- **on_failure** (*str*) – Behavior if no error is encountered in execution or post-processing but the result of the post-processing has turned the statement's *outcome* attribute to False, indicating the results returned by the statement have failed validation.
  - *None*: default behavior, exception will be raised
  - *c*: continue with execution

Returns (Statement): Statement object post-executing query.

outcome_txt (*self*, *_id: Optional[int] = None*) → str
Outcome as a string.

property outcome_html (*self*) → str
Outcome as an html admonition banner.

class snowmobile.core.**Column** (*original: str*, *current: Optional[str] = None*, *prior: Optional[str] = None*, *src: Optional[str] = None*)

Bases: *snowmobile.core.Generic*

A single column within a SnowFrame.

original
Original column name.

Type str

current
Current version of the column name.

Type str

prior
Prior version of the column name (version *n-1*).

Type str

src
Column source (original df or added by snowmobile).

Type str

original
Original column name.

Type str

src
Source of column; 'df' if from source DataFrame, 'snowmobile' otherwise.

Type str

current
Current column name.

Type str

prior
Prior (version of) column name.

Type str

update(*self*)
Migrate from prior to current context within this context.

lower(*self*) → str
Lower case column.

upper(*self*) → str
Upper case column.

static dedupe(*current: str*, *char: Optional[str] = None*) → str
Dedupes consecutive characters within a string.

Note:

- Must iterate through matches and perform replacements in the order of the **largest to the smallest by number of characters**; this is to avoid altering the matches found before replacing them.

Parameters

- **current** (*str*) – String containing characters to dedupe.
- **char** (*str*) – Character to dedupe.

reformat(*self*, *fill_char: Optional[str] = None*, *dedupe_special: bool = True*) → str
Reformat column for a load to the database.

Parameters

- **fill_char** (*str*) – Character to replace special characters and whitespace with; defaults to _.
- **dedupe_special** (*bool*) – Dedupe consecutive special characters; defaults to *True*.

class snowmobile.core.**Diff**(*sn: snowmobile.core.connection.Snowmobile = None*, ***kwargs*)
Bases: *snowmobile.core.qa.QA*

QA class for comparison of values within a table based on partitioning on a field.

partition_on
Column name to partition data on before comparing the partitioned datasets; defaults to 'src_description`.

Type str

end_index_at
Column name that marks the last column to use as an index column when joining the partitioned datasets back together.

Type str

compare_patterns
Regex patterns to match columns on that should be *included* in comparison (numeric columns you're running QA on).

Type list

ignore_patterns
Regex patterns to match columns on that should be *ignored* both for the comparison and the index.

Type list

generic_metric_col_nm
Column name to use for the melted field names; defaults to 'Metric'.

Type str

compare_cols
Columns that are used in comparison once statement is executed and parsing is applied.

Type list

drop_cols
Columns that are dropped once statement is executed and parsing is applied.

Type list

idx_cols
Columns that are used for the index to join the data back together once statement is executed and parsing is applied.

Type list

ub_raw
Maximum absolute raw difference (upper bound) that two fields that are being compared can differ from each other without causing a failure.

Type float

ub_perc
Maximum absolute percentage difference (upper bound) that two comparison fields can differ from each other without causing a failure.

Type float

Instantiates a `qa-diff` statement.

Parameters

- **delta_column_suffix** (*str*) – Suffix to add to columns that comparison is being run on; defaults to 'Delta'.
- **partition_on** (*str*) – Column to partition the data on in order to compare.
- **end_index_at** (*str*) – Column name that marks the last column to use as an index when joining the partitioned datasets back together.
- **compare_patterns** (*list*) – Regex patterns matching columns to be *included* in comparison.
- **ignore_patterns** (*list*) – Regex patterns to match columns on that should be *ignored* both for the comparison and the index.
- **generic_metric_col_nm** (*str*) – Column name to use for the melted field names; defaults to 'Metric'.

- **raw_upper_bound** (*float*) – Maximum absolute raw difference that two fields that are being compared can differ from each other without causing a failure.
- **percentage_upper_bound** (*float*) – Maximum absolute percentage difference that two comparison fields can differ from each other without causing a failure.

split_cols(*self*) → *snowmobile.core.qa.Diff*

Post-processes results returned from a `qa-diff` statement.

Executes private methods to split columns into:

- Index columns
- Drop columns
- Comparison columns

Then runs checks needed to ensure minimum requirements are met in order for a valid partition/comparison to be made.

property partitioned_by(*self*) → Set[Any]

Distinct values within the `partition_on` column that data is partitioned by.

static partitions_are_equal(*partitions: Dict[str, pd.DataFrame]*, *abs_tol: float*, *rel_tol: float*) → bool

Evaluates if a dictionary of DataFrames are identical.

Parameters

- **partitions** (*Dict[str, pd.DataFrame]*) – A dictionary of DataFrames returned by `snowmobile.DataFrame()`.
- **abs_tol** (*float*) – Absolute tolerance for difference in any value amongst the DataFrames being compared.
- **rel_tol** (*float*) – Relative tolerance for difference in any value amongst the DataFrames being compared.

Returns (bool): Indication of equality amongst all the DataFrames contained in `partitions`.

process(*self*) → *snowmobile.core.qa.Diff*

Post-processing for *Diff*-specific results.

class snowmobile.core.**Empty**(*sn:* snowmobile.core.connection.Snowmobile, ***kwargs*)

Bases: *snowmobile.core.qa.QA*

QA class for verification that a statement's results are empty.

The most widely applicable use of *Empty* is for simple verification that a table's dimensions are as expected.

Initialize self. See help(type(self)) for accurate signature.

process(*self*) → *snowmobile.core.qa.QA*

Over-ride method; checks if results are empty and updates outcome

class snowmobile.core.**SnowFrame**(*df: pandas.DataFrame*)

Bases: *snowmobile.core.Generic*

Extends a `DataFrame` with a `.snf` entry point.

shared_cols(*self*, *df2: pandas.DataFrame*) → List[Tuple[pd.Series, pd.Series]]

Returns list of tuples containing column pairs that are common between two DataFrames.

static series_max_diff_abs(*col1: pandas.Series*, *col2: pandas.Series*, *tolerance: float*) → bool

Determines if the max **absolute** difference between two `pandas.Series` is within a tolerance level.

static series_max_diff_rel(*col1: pandas.Series*, *col2: pandas.Series*, *tolerance: float*) → bool

Determines if the maximum **relative** difference between two `pandas.Series` is within a tolerance level.

df_max_diff_abs(*self*, *df2: pandas.DataFrame*, *tolerance: float*) → bool

Determines if the maximum **absolute** difference between any value in the shared columns of 2 DataFrames is within a tolerance level.

df_max_diff_rel(*self*, *df2: pandas.DataFrame*, *tolerance: float*) → bool

Determines if the maximum **relative** difference between any value in the shared columns of 2 DataFrames is within a tolerance level.

df_diff(*self*, *df2: pandas.DataFrame*, *abs_tol: Optional[float] = None*, *rel_tol: Optional[float] = None*) → bool

Determines if the column-wise difference between two DataFrames is within a relative **or** absolute tolerance level.

Note:

- `df1` and `df2` are assumed to have a shared, pre-defined index.
- Exactly **one** of `abs_tol` and `rel_tol` is expected to be a a valid float; the other is expected to be **None**.
- If valid float values are provided for both `abs_tol` and `rel_tol`, the outcome of the maximum **absolute** difference with respect to `abs_tol` will be returned regardless of the value of `rel_tol`.

Parameters

- **df2** (*pd.DataFrame*) – 2nd DataFrame for comparison.
- **abs_tol** (*float*) – Absolute tolerance; default is None.
- **rel_tol** (*float*) – Relative tolerance; default is None.

Returns (bool): Boolean indicating whether or not difference is within tolerance.

partitions(*self*, *on: str*) → Dict[str, pd.DataFrame]

Returns a dictionary of DataFrames given a DataFrame and a partition column.

Note:

- The number of distinct values within `partition_on` column will be 1:1 with the number of partitions that are returned.
- The `partition_on` column is dropped from the partitions that are returned.
- The depth of a vertical concatenation of all partitions should equal the depth of the original DataFrame.

Parameters **on** (*str*) – The column name to use for partitioning the data.

Returns (Dict[str, pd.DataFrame]): Dictionary of {(str) partition_value: (pd.DataFrame) associated subset of df}

ddl (*self*, *table: str*) → str
Returns a string containing 'create table' DDL given a table name

lower (*self*, *col: Optional[str] = None*) → pandas.DataFrame
Lower cases all column names **or** all values within *col* if pr.

upper (*self*, *col: Optional[str] = None*) → pandas.DataFrame
Upper cases all column names **or** all values within *col* if pr.

reformat (*self*)
Re-formats DataFrame's columns via `Column.reformat()`.

append_dupe_suffix (*self*)
Adds a trailing index number '_i' to duplicate column names.

to_list (*self*, *col: Optional[str] = None*, *n: Optional[int] = None*) → List
Succinctly retrieves a column as a list.

Parameters

- **col** (*str*) – Name of column.
- **n** (*int*) – Number of records to return; defaults to full depth of column.

add_tmstmp (*self*, *col_nm: Optional[str] = None*) → pandas.DataFrame
Adds a column containing the current timestamp to a DataFrame.

Parameters **col_nm** (*str*) – Name for column; defaults to *LOADED_TMSTMP*.

property original (*self*) → pandas.DataFrame
Returns the DataFrame in its original form (drops columns added by `SnowFrame` and reverts to original column names).

property has_dupes (*self*) → bool
DataFrame has duplicate column names.

cols_matching (*self*, *patterns: List[str]*, *ignore_patterns: List[str] = None*) → List[str]
Returns a list of columns given a list of patterns to find.

Parameters

- **patterns** (*List[str]*) – List of regex patterns to match columns on.
- **ignore_patterns** (*List[str]*) – Optional list of regex patterns to exclude.

Returns (List[str]): List of columns found/excluded.

cols_ending (*self*, *nm: str*, *ignore_patterns: Optional[List] = None*) → List[str]
Returns all columns up to `nm` in a DataFrame.

Parameters

- **nm** (*str*) – Name of column to end index at.
- **ignore_patterns** (*List[str]*) – Optional list of regex patterns to exclude in the list that's returned; primarily used to for getting *end-index-at* list while excluding *src_description*.

Returns (List[str]): List of column names matching criterion.

class snowmobile.core.**SQL**(*_query_func: Callable*, *_cfg: snowmobile.core.configuration.Configuration*, *nm: Optional[str] = None*, *schema: Optional[str] = None*, *obj: Optional[str] = None*, *auto_run: Optional[bool] = True*)

Bases: *snowmobile.core.Generic*

SQL class for generation & execution of common sql commands.

Intended to be interacted with as a parent of `Snowmobile`.

Note:

- All arguments except for `sn` are optional.
- The benefit of setting the other attributes on an instance of *SQL* is to (optionally) avoid passing the same information to multiple methods when generating a variety of statements around the same object.

---

nm
Object name to use in generated sql (e.g. 'some_table_name')

Type str

obj
Object type to use in generated sql (e.g. 'table')

Type str

schema
Schema to use when dot-prefixing sql; defaults to the schema with which the `sn` is connected to.

Type str

auto_run
Indicates whether to automatically execute the sql generated by a given method; defaults to *True*

Type bool

Initializes a `snowmobile.SQL` object.

info_schema(*self*, *loc: str*, *where: Optional[List[str]] = None*, *fields: Optional[List[str]] = None*, *order_by: Optional[List] = None*, *run: Optional[bool] = None*) → Union[str, pd.DataFrame]
Generic case of selecting from information schema location.

table_info(*self*, *nm: Optional[str] = None*, *fields: List[str] = None*, *restrictions: Dict[str, str] = None*, *order_by: List[Optional[str, int]] = None*, *all_schemas: bool = False*, *run: Optional[bool] = None*) → Union[str, pd.DataFrame]
Query `information_schema.tables` for a given table or view.

Parameters

- **nm** (*str*) – Table name, including schema if creating a stage outside of the current schema.
- **fields** (*List[str]*) – List of fields to include in returned results (e.g. ['table_name', 'table_type', 'last_altered'])
- **restrictions** (*List[str]*) – List of conditionals typed as literal components of a *where* clause (e.g. ["table_type = 'base table'", 'last_altered::date = current_date()']).
- **order_by** (*List[str]*) – List of fields or their ordinal positions to order the results by.

- **all_schemas** (*bool*) – Include tables/views from all schemas; defaults to *False*.
- **run** (*bool*) – Determines whether to run the generated sql or not; defaults to *None* which will reference the current value of the *auto_run* attribute which defaults to *True*.

Returns (Union[str, pd.DataFrame]):

Either:

1. The results of the query as a `pandas.DataFrame`, or
2. The generated query as a `str` of sql.

column_info(*self*, *nm: Optional[str] = None*, *fields: Optional[List] = None*, *restrictions: Optional[Dict] = None*, *order_by: Optional[List] = None*, *all_schemas: bool = False*, *run: Optional[bool] = None*) → Union[str, pd.DataFrame]

Query `information_schema.columns` for a given table or view.

Parameters

- **nm** (*str*) – Table name, including schema if creating a stage outside of the current schema.
- **fields** (*List[str]*) – List of fields to include in returned results (e.g. ['ordinal_position', 'column_name', 'data_type'])
- **restrictions** (*List[str]*) – List of conditionals typed as literal components of a *where* clause (e.g.["regexp_count(lower(column_name), 'tmstmp') = 0"]).
- **order_by** (*List[str]*) – List of fields or their ordinal positions to order the results by.
- **all_schemas** (*bool*) – Include tables/views from all schemas; defaults to *False*.
- **run** (*bool*) – Determines whether to run the generated sql or not; defaults to *None* which will reference the current value of the *auto_run* attribute which defaults to *True*.

Returns (Union[str, pd.DataFrame]):

Either:

1. The results of the query as a `pandas.DataFrame`, or
2. The generated query as a `str` of sql.

columns(*self*, *nm: Optional[str] = None*, *from_info_schema: bool = False*, *lower: bool = False*, *run: Optional[bool] = None*) → Union[str, List]

Returns an ordered list of columns for a table or view.

Note:

- Default behavior is to retrieve the columns for a table or view by selecting a single sample record and returning the column index from the DataFrame that's returned which is much faster than selecting the **column_names** from `information_schema.columns` pulling column names from the information schema
- This can be changed by passing *from_info_schema=True*.

Parameters

- **nm** (*str*) – Name of table or view, including schema if the table or view is outside of the current schema.
- **from_info_schema** (*bool*) – Indicates whether to retrieve columns via the `information_schema.columns` or by selecting a sample record from the table or view; defaults to *False*.
- **lower** (*bool*) – Lower case each column in the list that's returned.
- **run** (*bool*) – Execute generated sql; defaults to *True*, otherwise returns sql as a string.

Returns (Union[str, List]):

Either:

1. An ordered list of columns for the table or view, **or**
2. The query against the table or view as a `str` of sql.

select (*self*, *nm: Optional[str] = None*, *fields: Optional[List[str]] = None*, *apply: Optional[List[Tuple[str, str]]] = None*, *n: Optional[int] = None*, *run: Optional[bool] = None*, ***kwargs*) → Union[str, pd.DataFrame]

Generic *select* statement.

Parameters

- **nm** (*str*) – Table to select from, including schema if the table is outside of the current schema
- **fields** (*Optional[List[str]]*) – Select these fields (optional).
- **apply** (*Optional[List[Tuple[str, str]]]*) – Select aggregations of these fields.

  apply [(*this_func, to_this_field, [as_alias]*), (*.., .., [..]*),

  ]

  - `apply` should be provided as a list of tuples, each containing a minimum of 2 items (respectively) representing the aggregate function to apply and the field to which it should be applied
  - By default, the aggregated result inherits the name of the field being aggregated, including any qualifier (optionally) provided with the field name or an explicit alias included as a 3rd item within the tuple

    The following snippet exhaustively illustrates the functionality described above

```
sn.select(
   nm='sandbox.sample_table',
   apply=[
        ('count', 'col1'),
        ('count', 'distinct col1'),
        ('count', 'distinct col1', 'col1_dst'),
   ],
   run=False,
)
>>>
select
```

(continues on next page)

(continued from previous page)

```
 count(col1)
   as col1
 ,count(distinct col1)
   as distinct_col1
 ,count(distinct col1)
   as col1_dst
from sandbox.sample_table
```

- **n** (*int*) – Number of records to return, implemented as a 'limit' clause in the query; defaults to 1.
- **run** (*bool*) – Execute generated sql; defaults to *True*, otherwise returns sql as a string.

Returns (Union[str, pd.DataFrame]): Either:

1. The results of the query as a `pandas.DataFrame`, or
2. The generated query as a `str` of sql.

exists (*self*, *nm: Optional[str] = None*) → bool

Checks the existence of a table or view.

Parameters **nm** (*str*) – Name of table or view, including schema if the table or view is outside of the current schema.

Returns (bool): Boolean indication of whether or not the table or view exists.

is_distinct (*self*, *nm: Optional[str] = None*, *field: Optional[str] = None*) → bool

Checks if table *nm* is distinct on column *on_col*

Parameters

- **nm** (*str*) – Table name.
- **field** (*str*) – Column name.

count (*self*, *nm: Optional[str] = None*, *of: Optional[str] = None*, *dst_of: Optional[str] = None*, *as_perc: Optional[bool] = None*, *run: Optional[bool] = None*) → Union[int, float]

Number of records within a table or view.

Parameters

- **nm** (*str*) – Table name, including schema if querying outside current schema.
- **of** (*str*) – Column name (indistinct).
- **dst_of** (*str*) – Column name (distinct).
- **as_perc** (*bool*) – Option to return distinct count of the *dst_of* column as a percentage of the namespace depth of the table or view.
- **run** (*bool*) – Execute generated sql; defaults to *True*, otherwise returns sql as a string.

Returns (Union[str, pd.DataFrame]):

Either:

1. The results of the query as a `pandas.DataFrame`, or
2. The generated query as a `str` of sql.

show(*self*, *obj: str*, *in_loc: Optional[str] = None*, *names: bool = False*, *run: Optional[bool] = None*, ***kwargs*) → Union[pd.DataFrame, List[str], str]
Show schema objects of typ 'obj', optionally 'in_loc'.

Parameters

- **obj** (*str*) – Schema object type ('tables', 'file formats', etc).
- **in_loc** (*str*) – Snowflake location ('in schema sandbox', 'in database prod', etc).
- **names** (*bool*) – Return a list of schema object names only ('name' field).
- **run** (*bool*) – Execute the generated sql or return it as a string.

Returns (Union[pd.DataFrame, str]):

Either:

1. The results of the query as a `pandas.DataFrame`
2. The 'names' column of the results returned as a list
3. The generated query as a `str` of sql

ddl(*self*, *nm: Optional[str] = None*, *obj: Optional[str] = None*, *run: Optional[bool] = None*) → str
Query the DDL for an schema object.

Parameters

- **nm** (*str*) – Name of the object to get DDL for, including schema if object is outside of the current schema.
- **obj** (*str*) – Type of object to get DDL for (e.g. 'table', 'view', 'file-format').
- **run** (*bool*) – Execute generated sql; defaults to *True*, otherwise returns sql as a string.

Returns (str):

Either:

1. The results of the query as a `pandas.DataFrame`, or
2. The generated query as a `str` of sql.

comment(*self*, *nm: Optional[str] = None*, *obj: Optional[str] = None*, *set_as: Optional[str] = None*, *from_json: bool = False*, *as_json: bool = False*, *run: Optional[bool] = None*, ***kwargs*) → Union[str, Dict]
Get or set comment on a schema object.

Parameters

- **nm** (*str*) – Name of the schema object, including schema prefix if object is outside implicit scope of the current connection.
- **obj** (*str*) – Type of schema object (e.g. 'table', 'schema', etc).
- **set_as** (*str*) – Content to set as comment on schema object.
- **from_json** (*bool*) – Parse schema object comment as a string of json and return it as a dictionary.
- **as_json** (*bool*) – Dump contents of 'set_as' to a string of json prior to setting comment.

- **run** (*bool*) – Execute generated sql; defaults to *True*, otherwise returns sql as a string.
- ****kwargs** – Keyword argument to pass to *json.loads(comment)* if *from_json=True*.

Returns (Union[str, pd.DataFrame]):

Either:

1. The schema object comment as a `str`
2. The generated query as a `str` of sql.
3. The schema object comment as a dictionary if *from_json=True*

last_altered(*self*, *nm: Optional[str] = None*, *run: Optional[bool] = None*) → Union[str, pd.Timestamp]

Last altered timestamp for a table or view.

Parameters

- **nm** (*str*) – Table name, including schema if creating a stage outside of the current schema.
- **run** (*bool*) – Execute generated sql; defaults to *True*, otherwise returns sql as a string.

Returns (Union[str, pd.DataFrame]):

Either:

1. The results of the query as a `pandas.DataFrame`, or
2. The generated query as a `str` of sql.

truncate(*self*, *nm: Optional[str] = None*, *run: Optional[bool] = None*) → Union[str, pd.DataFrame]

Truncate a table.

Parameters

- **nm** (*str*) – Name of table, including schema if the table is outside of the current schema.
- **run** (*bool*) – Execute generated sql; defaults to *True*, otherwise returns sql as a string.

Returns (Union[str, pd.DataFrame]):

Either:

1. The results of the query as a `pandas.DataFrame`, or
2. The generated query as a `str` of sql.

drop(*self*, *nm: Optional[str] = None*, *obj: Optional[str] = None*, *run: Optional[bool] = None*) → Union[str, pd.DataFrame]

Drop a `Snowflake` object.

Parameters

- **nm** (*str*) – Schema object's name.
- **obj** (*str*) – Type of schema object (e.g. 'table', 'view', or 'schema')

- **run** (*bool*) – Execute generated statement; defaults to *True*, otherwise returns sql as a string.

Returns (Union[str, pd.DataFrame]):

Either:

1. The results of the query as a `pandas.DataFrame`, or
2. The generated query as a `str` of sql.

clone(*self*, *nm: Optional[str] = None*, *to: Optional[str] = None*, *obj: Optional[str] = None*, *run: Optional[bool] = None*, *replace: bool = False*) → Union[str, pd.DataFrame]

Clone a `Snowflake` object.

Warning:

- Make sure to read Snowflake's documentation for restrictions and considerations when cloning objects.

Note:

- In this specific method, the value provided to `nm` and `to` can be a single object name, a single schema, or both in the form of *obj_schema.obj_name* depending on the desired outcome.
- Additionally, **at least one of the** `nm` **or** `to` **arguments must be pr**.
- The defaults for the target object are constructed such that users can **either**:
  1. Clone objects to *other* schemas that inherit the source object's *name* without specifying so in the `to` argument, **or**
  2. Clone objects within the *current* schema that inherit the source object's *schema* without specifying so in the `to` argument.
- If providing a schema without a name to either argument, prefix the value provided with __ to signify it's a schema and not a lower-level object to be cloned.
  - e.g. providing *nm='sample_table'* and *to='__sandbox'* will clone *sample_table* from the current schema to *sandbox.sample_table*.
- An assertion error will be raised raised if neither argument is specified as *this would result in a command to clone an object and store it in an object that has the same name & schema as the object being cloned*.

Parameters

- **nm** (*str*) – Name of the object to clone, including schema if cloning an object outside of the current schema.
- **to** (*str*) – Target name for cloned object, including schema if cloning an object outside of the current schema.
- **obj** (*str*) – Type of object to clone (e.g. 'table', 'view', 'file-format'); defaults to *table*.
- **run** (*bool*) – Execute generated sql; defaults to *True*, otherwise returns sql as a string.

- **replace** (*bool*) – Indicates whether to replace an existing stage if pre-existing; default is *False*.

Returns (Union[str, pd.DataFrame]):

Either:

1. The results of the query as a `pandas.DataFrame`, or
2. The generated query as a `str` of sql.

create_stage(*self*, *nm_stage: str*, *nm_format: str*, *replace: bool = False*, *run: Optional[bool] = None*) → Union[str, pd.DataFrame]

Create a staging table.

Parameters

- **nm_stage** (*str*) – Name of stage to create, including schema if creating a stage outside of the current schema.
- **nm_format** (*str*) – Name of file format to specify for the stage, including schema if using a format from outside of the current schema.
- **run** (*bool*) – Execute generated sql; defaults to *True*, otherwise returns sql as a string.
- **replace** (*bool*) – Indicates whether to replace an existing stage if pre-existing; default is *False*.

Returns (Union[str, pd.DataFrame]):

Either:

1. The results of the query as a `pandas.DataFrame`, or
2. The generated query as a `str` of sql.

put_file_from_stage(*self*, *path: Union[Path, str]*, *nm_stage: str*, *options: Optional[Dict] = None*, *ignore_defaults: bool = False*, *run: Optional[bool] = None*) → Union[str, pd.DataFrame]

Generates a 'put' command into a staging table from a local file.

Parameters

- **path** (*Union[Path, str]*) – Path to local data file as a `pathlib.Path` or string.
- **nm_stage** (*str*) – Name of the staging table to load into.
- **run** (*bool*) – Execute generated sql; defaults to *True*, otherwise returns sql as a string.
- **options** (*dict*) – Optional arguments to add to *put* statement in addition to the values specified in the `loading.put` section of **snowmobile.toml**.
- **ignore_defaults** (*bool*) – Option to ignore the values specified in **snowmobile.toml**; defaults to *False*.

Returns (Union[str, pd.DataFrame]):

Either:

1. The results of the query as a `pandas.DataFrame`, or

2. The generated query as a `str` of sql.

copy_into_table_from_stage(*self*, *nm: str*, *nm_stage: str*, *options: Optional[Dict] = None*, *ignore_defaults: bool = False*, *run: Optional[bool] = None*) → Union[str, pd.DataFrame]

Generates a command to copy data into a table from a staging table.

Parameters

- **nm** (*str*) – Name of the object to drop, including schema if creating a stage outside of the current schema.
- **nm_stage** (*str*) – Name of the staging table to load from.
- **run** (*bool*) – Execute generated sql; defaults to *True*, otherwise returns sql as a string.
- **options** (*dict*) – Optional arguments to add to *put* statement in addition to the values specified in the `loading.put` section of **snowmobile.toml**.
- **ignore_defaults** (*bool*) – Option to ignore the values specified in **snowmobile.toml**; defaults to *False*.

Returns (Union[str, pd.DataFrame]):

Either:

1. The results of the query as a `pandas.DataFrame`, or
2. The generated query as a `str` of sql.

current(*self*, *obj: str*, *run: Optional[bool] = None*) → Union[str, Union[str, int]]

Generic implementation of 'select current' for session-based objects.

Parameters

- **obj** (*str*) – Type of object to retrieve information for (schema, session, ..).
- **run** (*bool*) – Execute generated sql; defaults to *True*, otherwise returns sql as a string.

Returns (Union[str, pd.DataFrame]):

Either:

1. The results of the query as a `pandas.DataFrame`, or
2. The generated query as a `str` of sql.

current_session(*self*, *run: Optional[bool] = None*) → Union[str, pd.DataFrame]

Select the current session.

current_schema(*self*, *run: Optional[bool] = None*) → Union[str, pd.DataFrame]

Select the current schema.

current_database(*self*, *run: Optional[bool] = None*) → Union[str, pd.DataFrame]

Select the current database.

current_warehouse(*self*, *run: Optional[bool] = None*) → Union[str, pd.DataFrame]

Select the current warehouse.

current_role(*self*, *run: Optional[bool] = None*) → Union[str, pd.DataFrame]

Select the current role.

use(*self*, *obj: str*, *nm: str*, *run: Optional[bool] = None*)
Generic implementation of 'use' command for schema objects.

Parameters

- **nm** (*str*) – Name of object to use (schema name, warehouse name, role name, ..).
- **obj** (*str*) – Type of object to use (schema, warehouse, role, ..).
- **run** (*bool*) – Execute generated sql; defaults to *True*, otherwise returns sql as a string.

Returns (Union[str, pd.DataFrame]):

Either:

1. The results of the query as a `pandas.DataFrame`, or
2. The generated query as a `str` of sql.

use_schema(*self*, *nm: Optional[str] = None*, *run: Optional[bool] = None*) → Union[str, pd.DataFrame]
Use schema command.

use_database(*self*, *nm: Optional[str] = None*, *run: Optional[bool] = None*) → Union[str, pd.DataFrame]
Use database command.

use_warehouse(*self*, *nm: Optional[str] = None*, *run: Optional[bool] = None*) → Union[str, pd.DataFrame]
Use warehouse command.

use_role(*self*, *nm: Optional[str] = None*, *run: Optional[bool] = None*) → Union[str, pd.DataFrame]
Use role command.

static order(*by: List[Union[int, str]]*) → str
Generates 'order by' clause from a list of fields or field ordinal positions.

static where(*restrictions: Dict*) → str
Generates a 'where' clause based on a dictionary of restrictions.

Parameters **restrictions** (*dict*) – A dictionary of conditionals where each key/value pair respectively represents the left/right side of a condition within a 'where' clause.

Returns (str): Formatted where clause.

static fields(*fields: Optional[List[str]] = None*) → str
Utility to generate fields within a 'select' statement.

class snowmobile.core.**Markup**(*sn: snowmobile.core.connection.Snowmobile*, *path: pathlib.Path*, *contents: Dict[int, Union[Statement, Marker]]*, *nm: Optional[str] = None*, *prefix: Optional[str] = None*, *suffix: Optional[str] = None*, *root_dir: Optional[Union[str, Path]] = None*, *sub_dir: Optional[str] = None*, *incl_sql: bool = True*, *incl_markers: bool = True*, *incl_sql_tag: bool = False*, *incl_exp_ctx: bool = True*, *result_wrap: Optional[str] = None*)
Bases: *snowmobile.core.Generic*

Contains all sections within the context of a `Script`.

Parameters

- **sn** (`Snowmobile`) – A `Snowmobile` instance.

- **path** (*Path*) – A full path to the sql file that script was instantiated from.
- **contents** (*Dict[int, Union[Statement, Marker]]*) – A dictionary of the script's contents (st and markers) by index position.
- **nm** (*Optional[str]*) – Alternate file name to use; defaults to `path.name`.
- **prefix** (*Optional[str]*) – Prefix to prepend to original file name when exporting.
- **suffix** (*Optional[str]*) – Suffix to append to original file name when exporting.
- **root_dir** (*Optional[Union[str, Path]]*) – Alternate target directory for exports; defaults to `./.snowmobile` where `.` is the directory containing the sql file that the `script` was created from.
- **sub_dir** (*Optional[str]*) – Alternate sub-directory name; defaults to `path.name` where `path` is a full `Path` to the sql file that the `script` was created from.
- **incl_sql** (*bool*) – Include statements in export.
- **incl_markers** (*bool*) – Include markers in export.
- **incl_sql_tag** (*bool*) – Include the raw wrap in the sql that is rendered in the *md* export.
- **incl_exp_ctx** (*bool*) – Include (configurable) disclaimer at the top of exported *sql* file.

exported
List of file paths that current instance has exported to.

Type List[Path]

created
List of directory paths that current instance has created (should mostly apply for initial scaffolding build on first run only).

Type List[Path]

property export_dir(*self*) → pathlib.Path
Documentation sub-directory; *.snowmobile* by default.

property sections(*self*) → Dict[int, *Section*]
Dictionary of all *sections* by index position.

property markdown(*self*) → str
Full markdown file as a string.

property sql(*self*)
SQL for save.

save(*self*, *md: bool = True*, *sql: bool = True*) → None
Save files to disk.

Parameters

- **md** (*bool*) – Export a generated markdown file.
- **sql** (*bool*) – Export a generated sql file.

class snowmobile.core.**Script**(*sn: Optional[Snowmobile] = None*, *path: Optional[Path, str] = None*, *sql: Optional[str] = None*, *as_generic: bool = False*, *delay: bool = True*, ***kwargs*)
Bases: *snowmobile.core.Generic*

Parser and operator of local sql files.

Parameters

- **sn** (snowmobile.core.connection.Snowmobile) – An instance of *Snowmobile*.
- **path** (*Optional[str]*) – A full path to a sql file or readable text file containing valid sql code.
- **path** – A raw string of valid sql code as opposed to reading from a path.
- **as_generic** (*bool*) – Instantiate all statements as generic st; skips all checks for a mapping of a statement anchor to a derived statement class to instantiate in the place of a generic *Statement*.
- **delay** (*bool*) – Delay connection of the *Snowmobile*; only applicable if the sn argument is omitted and *Script* is instantiating a *Snowmobile* in its absence.
- ****kwargs** – Any keyword arguments to pass to *Snowmobile*; only applicable if the sn argument is omitted and *Script* is instantiating a *Snowmobile* in its absence

sn

An instance of *Snowmobile*

Type *snowmobile.core.connection.Snowmobile*

patterns

Configured patterns from *snowmobile.toml*.

Type *snowmobile.core.cfg.script.Pattern*

as_generic

Instantiate all statements as generic st; skips all checks for a mapping of a statement anchor to a derived statement class to instantiate in the place of a generic *Statement*.

Type bool

filters

Dictionary of filters that have been passed to the current instance of *snowmobile.core.Script*.

Type Dict[Any[str, int], Dict[str, Set]]

markers

Dictionary of all markers found in the script.

Type Dict[int, *cfg.Marker*]

path

Path to sql file (e.g. *full/path/to/script.sql*).

Type Path

name

Name of sql file (e.g. *script.sql*).

Type str

source

Raw sql text of script; will be the text contained in the raw sql file when initially read from source and reflect any modifications to the script's contents made post-instantiation.

Type str

read(*self*, *path: pathlib.Path = None*) → *snowmobile.core.script.Script*

Runs quick path validation and reads in a sql file as a string.

A valid *path* must be provided if the *script.path* attribute hasn't been set; `ValueErrors` will be thrown if neither is valid.

Parameters **path** (*pathlib.Path*) – Full path to a sql object.

from_str(*self*, *sql: str*, *name: str*, *directory: pathlib.Path = Path.cwd()*) → *snowmobile.core.script.Script*
Instantiates a raw string of sql as a script.

source(*self*, *original: bool = False*) → str
The script's sql as a raw string.

parse_one(*self*, *s: Union[sqlparse.sql.Statement, str]*, *index: Optional[int] = None*, *nm: Optional[str] = None*) → None
Adds a statement object to the script.

Default behavior will only add `sqlparse.sql.Statement` objects returned from `script.source_stream`.

`clean_parse()` utility function is utilized so that generated sql within Python can be inserted back into the script as raw strings.

Parameters

- **s** (*Union[sqlparse.sql.Statement, str]*) – A sqlparse.sql.Statement object or a raw string of SQL for an individual statement.
- **index** (*int*) – Index position of the statement within the script; defaults to `n + 1` if index is not provided where `n` is the number of statements within the script at the time `parse_one()` is called.
- **nm** (*Optional[str]*) – Optionally provided the name of the statement being added; the script instance will treat this value as if it were provided within an in-script wrap.

parse_stream(*self*, *stream: str*) → None
Parses a stream of sql and adds onto existing Script contents.

filter(*self*, *incl_kw: Optional[List[str], str] = None*, *incl_obj: Optional[List[str], str] = None*, *incl_desc: Optional[List[str], str] = None*, *incl_anchor: Optional[List[str], str] = None*, *incl_nm: Optional[List[str], str] = None*, *excl_kw: Optional[List[str], str] = None*, *excl_obj: Optional[List[str], str] = None*, *excl_desc: Optional[List[str], str] = None*, *excl_anchor: Optional[List[str], str] = None*, *excl_nm: Optional[List[str], str] = None*, *as_id: Optional[Union[str, int]] = None*, *from_id: Optional[Union[str, int]] = None*, *last: bool = False*) → ContextManager[*Script*]
Subset the script based on attributes of its st.

`script.filter()` returns a modified instance of script that can be operated on within the context defined.

Note: Keyword arguments beginning with `incl` or `excl` expect a string or a list of strings containing regex patterns with which to check for a match against the associated attribute of its st' `Name`.

Parameters

- **incl_kw** – Include only `kw`
- **incl_obj** – Include only `obj`
- **incl_desc** – Include only `desc`

- **incl_anchor** – Include only `anchor`
- **incl_nm** – Include only `nm`
- **excl_kw** – Exclude `kw`
- **excl_obj** – Exclude `obj`
- **excl_desc** – Exclude `desc`
- **excl_anchor** – Exclude `anchor`
- **excl_nm** – Exclude `nm`
- **as_id** – ID to assign the filters passed to method; used to populated the *`filters`* attribute
- **from_id** – ID previously used on the same instance of *`Script`* from which to populate filtered arguments
- **last** – Re-use the last set of filters passed to context manager.

Returns (Script): The instance of script based on the context imposed by arguments pr.

property depth (*self*) → int
Count of statements in the script.

property lines (*self*) → int
Number of lines in the script

property excluded (*self*)
All statements by index position excluded from the current context.

property executed (*self*) → Dict[int, *Statement*]
Executed statements by index position included in the current context.

reset (*self*, *index: bool = False*, *ctx_id: bool = False*, *in_context: bool = False*, *scope: bool = False*, *_filter: bool = False*) → *snowmobile.core.script.Script*
Resets indices and scope on all statements to their state as read from source.

Invoked before exiting *`filter()`* context manger to reverse the revised indices set by `index_to()` and inclusion/ exclusion scope set by `Statement.Name.scope()`.

property duplicates (*self*) → Dict[str, int]
Dictionary of indistinct statement names/tags within script.

s (*self*, *_id: Optional[str, int] = None*) → Any[*Statement*, *Empty*, *Diff*]
Fetch a single statement by _id.

property st (*self*) → Dict[Union[int, str], *Statement*]
Accessor for all statements.

dtl (*self*, *full: bool = False*, *excluded: bool = False*, *title: bool = True*, *r: bool = False*) → Union[str, None]
Prints summary of statements within the current scope to console.

property first_s (*self*)
First statement by index position.

property last_s (*self*)
Last statement by index position

property first (*self*) → Union[*Statement*, *Empty*, *Diff*]
First statement executed.

property last (*self*) → Union[*Statement*, *Empty*, *Diff*]
Last statement executed.

doc (*self*, *nm: Optional[str] = None*, *prefix: Optional[str] = None*, *suffix: Optional[str] = None*, *incl_markers: Optional[bool] = True*, *incl_sql: Optional[bool] = True*, *incl_exp_ctx: Optional[bool] = True*, *result_wrap: Optional[str] = None*) → *snowmobile.core.Markup*
Returns a `Markup` from the script.

Parameters

- **nm** (*Optional[str]*) – Alternate file name to use.
- **prefix** (*Optional[str]*) – Prefix for file name.
- **suffix** (*Optional[str]*) – Suffix for file name.
- **incl_markers** (*Optional[bool]*) – Include markers in exported files.
- **incl_sql** (*Optional[bool]*) – Include sql in exported files.
- **incl_exp_ctx** (*Optional[bool]*) – Include disclaimer of programmatic save in exported sql file.

Returns A `Markup` instance based on the contents included in the script's context.

ids (*self*, *_id: Optional[Union[Tuple, List]] = None*) → List[int]
Utility function to get a list of statement IDs given an *_id*.

Invoked within script.run() if the *_id* parameter is either a:

(1) tuple of integers (lower and upper bound of statement indices to run)

(2) list of integers or strings (statement names or indices to run)

(3) default=None; returns all statement indices within scope if so

Parameters **_id** (*Union[Tuple, List]*) – _id field provided to script.run() if it's neither an integer or a string.

Returns (List[int]): A list of statement indices to run.

run (*self*, *_id: Optional[str, int, Tuple[int, int], List] = None*, *as_df: bool = True*, *on_error: Optional[str] = None*, *on_exception: Optional[str] = None*, *on_failure: Optional[str] = None*, *lower: bool = True*, *render: bool = False*, ***kwargs*) → None
Performs statement-by-statement execution of the script's contents.

Executes script's contents that are included within its current context and any (optional) value passed to the `_id` argument.

Note: Keyword arguments `on_exception` and `on_failure` are only applicable to derived classes of `Statement` (e.g., those within `snowmobile.core.qa` by default).

Parameters

- **_id** (*Optional[str, int, Tuple[int, int], List]*) –

  Identifier for statement(s) to execute, can be either:

  - *None* (default); execute all statements
  - A single statement's *nm*
  - A single statement's index position

  - A tuple of lower/upper index bounds of statements to execute
  - A list of statement names or index positions to execute
- **as_df** (*bool*) – Store statement's results as a `DataFrame`; defaults to *True*
- **on_error** (*Optional[str]*) – Action to take on **execution** error; providing *c* will continue execution as opposed to raising exception.
- **on_exception** (*Optional[str]*) – Action to take on **post-processing** error from a derived *Statement*; providing *c* will continue execution as opposed to raising exception.
- **on_failure** (*Optional[str]*) – Action to take on **failure** of post-processing assertion from a derived *Statement*; providing *c* will continue execution as opposed to raising exception.
- **lower** (*bool*) – Lower-case columns in results returned if `as_df=True`.
- **render** (*bool*) – Render sql executed as markdown; only applicable in Jupyter/iPython environments.
- ****kwargs** –

items(*self*, *by_index: bool = True*, *ignore_scope: bool = False*, *statements: bool = True*, *markers: bool = False*, *validate: bool = True*) → ItemsView[Union[int, str], Union[*Statement*, Marker]]
Dunder items.

keys(*self*, ***kwargs*) → KeysView[Union[int, str]]
Access keys of items only.

values(*self*, ***kwargs*) → ValuesView[Union[int, str]]
Access values of items only.

dict(*self*, ***kwargs*) → Dict
Unpacking items view into an actual dictionary.

class snowmobile.core.**Table**(*df: pandas.DataFrame*, *table: str*, *sn: Optional[*Snowmobile*] = None*, *if_exists: Optional[str] = None*, *as_is: bool = False*, *path_ddl: Optional[Union[str, Path]] = None*, *path_output: Optional[str, Path] = None*, *file_format: Optional[str] = None*, *incl_tmstmp: Optional[bool] = None*, *tmstmp_col_nm: Optional[str] = None*, *reformat_cols: Optional[bool] = None*, *validate_format: Optional[bool] = None*, *validate_table: Optional[bool] = None*, *upper_case_cols: Optional[bool] = None*, *lower_case_table: Optional[bool] = None*, *keep_local: Optional[bool] = None*, *on_error: Optional[str] = None*, *check_dupes: Optional[bool] = None*, *load_copy: Optional[bool] = None*, ***kwargs*)

Bases: *snowmobile.core.Generic*

Constructed with a `DataFrame` and a table name to load into.

The `df` and `table`'s compatibility can be inspected prior to calling the `Table.load()` method or by providing *as_is=True*` when instantiating the object; the latter will kick off the loading process invoked by *.load()* based on the parameters provided to *snowmobile.Table()*.

Parameters

- **df** (*DataFrame*) – The `DataFrame` to load.
- **table** (*str*) – The table name to load `df` into.

- **sn** (*Optional[Snowmobile]*) – An instance of `Snowmobile`; can be used to load a table on a specific connection or from a specific `snowmobile.toml` file.
- **if_exists** (*Optional[str]*) – Action to take if `table` already exists - options are *fail*, *replace*, *append*, and *truncate*; defaults to `append`.
- **as_is** (*bool*) – Load `df` into `table` based on the parameters provided to *Table* without further pre-inspection by the user; defaults to *False*.
- **path_ddl** (*Optional[Path]*) – Alternate path to file format DDL to use for load.
- **keep_local** (*Optional[bool]*) – Keep local file that is written out as part of the bulk loading process; defaults to *False*.
- **path_output** (*Optional[str Path]*) – Path to write output local file to; defaults to a generated file name exported in the current working directory.
- **file_format** (*Optional[str]*) – The name of the file_format to use when loading `df`; defaults to `snowmobile_default_psv`.
- **incl_tmstmp** (*Optional[bool]*) – Include timestamp of load as part of `table`; defaults to *True*.
- **tmstmp_col_nm** (*Optional[str]*) – Name to use for load timestamp if `incl_tmstmp=True`; defaults to *loaded_tmstmp*.
- **upper_case_cols** (*Optional[bool]*) – Upper case columns of `df` when loading into `table`; defaults to *True*.
- **reformat_cols** (*Optional[bool]*) – Reformat applicable columns of `df` to be DB-compliant; defaults to *True*.

  Reformatting primarily entails:

  - Replacing spaces and special characters with underscores
  - De-duping consecutive special characters
  - De-duping repeated column names; adds an `_i` suffix to duplicate fields where `i` is the nth duplicate name for a field

- **validate_format** (*Optional[bool]*) – Validate the file format being used prior to kicking off the load; defaults to *True*.

  Validation entails:

  - Checking if the file format being used already exists based on formats accessible to the current connection
  - Executing DDL for the file format being used if not, pulled from the `DDL` *ext-location* and the statement name `create file format~{format name}`

  Tip: Providing *validate_format=False* will speed up loading time when batch-loading into an existing table by skipping this step

- **validate_table** (*Optional[bool]*) – Perform validations of `df` against `table` prior to kicking off the loading process; defaults to *True*.

  Validation entails:

  - Checking the existence of `table`; no further validation is performed if it does **not** exist

- Compares the columns of `df` to the columns of `table` and stores results for use during loading process

---

Note: Table validation results are used in conjunction with the `if_exists` parameter to determine the desired behavior based on the (potential) existence of `table` and its compatibility with `df`.

Tip: Providing *validate_table=False* will speed up loading time time when batch-loading into an existing table

---

- **lower_case_table** (*Optional[bool]*) – Lower case `table` name; defaults to *False*.
- **on_error** (*Optional[str]*) – Action to take if an exception is encountered as part of the validating or loading process - providing `on_error='c'` will *continue* past an exception as opposed to raising it; defaults to *None* meaning any exception encountered will be raised
- **check_dupes** (*Optional[bool]*) – Check for duplicate field names in `df`; defaults to *True*.
- **load_copy** (*Optional[bool]*) – Alter and load a deep copy of `df` as opposed to the `df` in-memory as passed to the parameter; defaults to *True*.

db_responses
Responses from database during loading process.

Type Dict[str, str]

loaded
Table was loaded successfully.

Type bool

load(*self*, *if_exists: Optional[str] = None*, *from_script: pathlib.Path = None*, *verbose: bool = True*, ***kwargs*) → *snowmobile.core.table.Table*
Loads `df` into `table`.

Parameters

- **if_exists** (*Optional[str]*) – Determines behavior to take if the table being loaded into already exists; defaults to **append**; options are **replace**, **append**, **truncate**, and **fail**
- **from_script** (*Optional[Union[Path, str]]*) – Path to sql file containing custom DDL for `table`; DDL is assumed to have a valid statement name as is parsed by *Script* and following the naming convention of `create table~TABLE` where `TABLE` is equal to the value provided to the `table` keyword argument
- **verbose** (*bool*) – Verbose console output; defaults to **True**

Returns (Table): The *Table* after attempting load of `df` into `table`; a successful load can be verified by inspecting *loaded*

property exists(*self*) → bool
Indicates if the target table exists.

col_diff(*self*, *mismatched: bool = False*) → Dict[int, Tuple[str, str]]
Returns diff detail of local DataFrame to in-warehouse table.

property cols_match(*self*) → bool
Indicates if columns match between DataFrame and table.

load_statements(*self*, *from_script: pathlib.Path*) → List[str]
Generates exhaustive list of the statements to execute for a given instance of loading a DataFrame.

to_local(*self*, *quote_all: bool = True*) → None
Export to local file via configuration in `snowmobile.toml`.

property tm_load(*self*) → int
Seconds elapsed during loading.

property tm_validate_load(*self*) → int
Seconds elapsed during validation.

property tm_total(*self*) → int
Total seconds elapsed for load.

validate(*self*, *if_exists: str*) → None
Validates load based on current state through a variety of operations.

Parameters **if_exists** (*str*) – Desired behavior if table already exists; intended to be passed in from `table.load()` by default.

CHAPTER

EIGHT

SNIPPETS

This is a generated reference page for complete code snippets used throughout the rest of the documentation.

8.1 Configuration

8.1.1 *inspect_configuration.py*

Download

text

```
"""
Instantiate a delayed snowmobile.Snowmobile object and inspect configuration model.
../docs/snippets/configuration/inspect_configuration.py
"""
import snowmobile

sn = snowmobile.Snowmobile(delay=True)

type(sn.cfg)               #> snowmobile.core.configuration.Configuration
print(sn.cfg.location)     #  'path/to/your/snowmobile.toml'

type(sn.cfg.connection)    #> snowmobile.core.cfg.connection.Connection
type(sn.cfg.loading)       #> snowmobile.core.cfg.loading.Loading
type(sn.cfg.script)        #> snowmobile.core.cfg.script.Script
type(sn.cfg.sql)           #> snowmobile.core.cfg.other.SQL
type(sn.cfg.ext_sources)   #> snowmobile.core.cfg.other.Location

# -- complete example; should run 'as is' --
```

8.2 Overview

8.2.1 *sample_table.sql*

Download

text

```sql
-- ..docs/snippets/getting_started/sample_table.sql

create or replace table sample_table (
        col1 number(18,0),
        col2 number(18,0)
);

insert into sample_table with
sample_data as (
  select
    uniform(1, 10, random(1)) as rand_int
  from table(generator(rowcount => 3)) v
)
  select
    row_number() over (order by a.rand_int) as col1
    ,(col1 * col1) as col2
  from sample_data a;

select * from sample_table;

/*-qa-empty~verify 'sample_table' is distinct on 'col1'-*/
select
  a.col1
  ,count(*)
from sample_table a
group by 1
having count(*) <> 1;

/*-insert into~any_other_table-*/
insert into any_other_table (
  select
    a.*
    ,tmstmp.tmstmp as insert_tmstmp
  from sample_table a
  cross join (select current_timestamp() as tmstmp)tmstmp
);
```

8.3 Script

8.3.1 *intro.sql*

Download

text

```sql
/*
../snippets/script/intro.sql
Demonstrate basic parsing functionality.
*/

/*-
__intro.sql__
__authored-by: Some Chap or Lass
```

(continues on next page)

(continued from previous page)

```
__authored-on: Some Day or Year
__p*_***:
**Impetus**: *SQL is older than time and isn't going anywhere; might we allow a␣
↪simple markup syntax?*
-*/

/*-
create table~sample_table; DDL
__description: This is an example statement description
-*/
create or replace table sample_table (
        col1 number(18,0),
        col2 number(18,0)
);
```

8.4 Intro

8.4.1 *intro1.sql*

Download

text

```
-- ./docs/snippets/script/intro/intro1.sql

create or replace table sample_table (
  col1 number(18,0),
  col2 number(18,0)
);

insert into sample_table (col1, col2) values(1, 2);

select * from sample_table;
```

8.4.2 *keyword_exceptions.sql*

Download

text

```
-- ..docs/snippets/script/keyword_exceptions.sql

-- kw = 'select'
select * from any_table;

-- kw = 'select'
with some_cte as (
 select * from any_table
)
 select * from some_cte;
```

8.4.3 *markup.sql*

Download

text

```
-- ..docs/snippets/script/markup.sql

/*-
__name: Example.sql
__description: This is an example description field.
-*/
```

8.4.4 *overview-base-sn.sql*

Download

text

```
-- ..docs/snippets/script/overview-base.sql

/*-
__overview-base-sn.sql__
__authored-by: some person
__authored-on: some date
__context*_***: This is a contrived example of how a script can be marked up and␣
↪parsed by Snowmobile.
-*/

/*-create table sample_table~DDL-*/
create or replace table sample_table (
  col1 number(18,0),
  col2 number(18,0),
  insert_tmstmp timestamp
);

/*-insert into~sample_table-*/
insert into sample_table with
sample_data as (
  select
    uniform(1, 10, random(1)) as rand_int
  from table(generator(rowcount => 3)) v
)
  select
    row_number() over (order by a.rand_int) as col1
    ,(col1 * col1) as col2
    ,tmstmp.tmstmp as insert_tmstmp
  from sample_data a
  cross join (select current_timestamp() as tmstmp) tmstmp;

/*-sample records~sample_table-*/
select
  *
```

(continues on next page)

(continued from previous page)

```
from sample_table;

/*-qa-empty~verify sample_table is distinct on col1-*/
select
        a.col1
        ,count(*)
from sample_table a
group by 1
having count(*) > 1;

/*-create table~any_other_table-*/
create or replace table any_other_table
clone sample_table;

/*-alter table~staged_tmstmp addition-*/
alter table any_other_table add column staged_tmstmp timestamp;

/*-insert into~any_other_table-*/
insert into any_other_table (
  select
    a.col1
    ,a.col2
    ,tmstmp.tmstmp
    ,a.insert_tmstmp
  from sample_table a
  cross join (select current_timestamp() as tmstmp) tmstmp
);

/*-qa-empty~verify any_other_table is distinct on col1-*/
select
        a.col1
        ,count(*)
from any_other_table a
group by 1
having count(*) > 1;

/*-truncate table~sample_table-*/
truncate table sample_table;
```

8.4.5 *overview-base.sql*

Download

text

```
-- ..docs/snippets/script/overview-base.sql

/*
 author: some person
   date: some date
context: this is a contrived example of what a messy sql file can look like
*/

```

(continues on next page)

(continued from previous page)

```
-- DDL: one-time execution
create or replace table sample_table (
  col1 number(18,0),
  col2 number(18,0),
  insert_tmstmp timestamp
);

-- update only
insert into sample_table with
sample_data as (
  select
    uniform(1, 10, random(1)) as rand_int
  from table(generator(rowcount => 3)) v
)
  select
    row_number() over (order by a.rand_int) as col1
    ,(col1 * col1) as col2
    ,tmstmp.tmstmp as insert_tmstmp
  from sample_data a
  cross join (select current_timestamp() as tmstmp) tmstmp;
-- select * from sample_table;

-- ensure distinct
-- select
--         a.col1
--         ,count(*)
-- from sample_table a
-- group by 1
-- having count(*) > 1;

-- select * from some_random_table_that_no_longer_matters;

-- clone stage
create or replace table any_other_table
clone sample_table;

-- add original tmstmp
alter table any_other_table add column staged_tmstmp timestamp;

-- insert data
insert into any_other_table (
  select
    a.col1
    ,a.col2
    ,tmstmp.tmstmp
    ,a.insert_tmstmp
  from sample_table a
  cross join (select current_timestamp() as tmstmp) tmstmp
);

-- ensure distinct
-- select
--         a.col1
--         ,count(*)
-- from any_other_table a
-- group by 1
-- having count(*) > 1;
```

(continues on next page)

(continued from previous page)

```

-- compare final table to staged values
--          select * from sample_table a
-- union all
--          select a.col1, a.col2, a.staged_tmstmp from any_other_table a;

-- truncate staging table
truncate table sample_table;
```

8.4.6 *overview-statement-intro.py*

Download

text

```
"""
Instantiate `script` from 'overview.sql' and inspect high-level contents.
../docs/snippets/script/overview-base-parsing.py
"""

# Setup ------------------------------------------------------------------------
from pathlib import Path
paths = {p.name: p for p in Path.cwd().glob('**/*.sql')}
path = paths['overview.sql']

import snowmobile


# Example ----------------------------------------------------------------------

# -- Block 1 --
script = snowmobile.Script(path=path)
script.dtl()
"""
>>>
overview.sql
============
1: Statement('create table~s1')
2: Statement('insert into~s2')
3: Statement('select data~s3')
4: Statement('select all~sample_table')
5: Statement('create transient table~s5')
6: Statement('insert into~s6')
7: Statement('drop table~s7')
"""

# -- Block 2 --
# Store a few st, accessed by index position
s_first, s_last = script(1), script(-1)

# first sql keyword
print(s_first.kw)  #> create
print(s_last.kw)   #> drop
```

(continues on next page)

(continued from previous page)

```

# position within `script`
print(s_first.index)  #> 1
print(s_last.index)   #> 7

# -- Block 3 --
script.run(1)     # .run() from `script`
script(1).run()   # .run() from `statement`

# -- Block 4 --
# `script` details as read from 'overview.sql'
print(script.depth)   #> 7
print(script(1).nm)   #> create table~s1
print(script(-1).nm)  #> drop table~s7

with script.filter(excl_kw=['select', 'drop']) as s:
    print(s.depth)   #> 4
    print(s(1).nm)   #> create table~s1
    print(s(-1).nm)  #> insert into~s4
    s.dtl()
"""
>>>
overview.sql
============
1: Statement('create table~s1')
2: Statement('insert into~s2')
3: Statement('create transient table~s3')
4: Statement('insert into~s4')
"""
```

8.4.7 *overview.sql*

Download

text

```
-- ..docs/snippets/script/overview.sql

create or replace table sample_table (
        col1 number(18,0),
        col2 number(18,0)
);

insert into sample_table with
sample_data as (
  select
    uniform(1, 10, random(1)) as rand_int
  from table(generator(rowcount => 3)) v
)
  select
    row_number() over (order by a.rand_int) as col1
    ,(col1 * col1) as col2
  from sample_data a;
```

(continues on next page)

(continued from previous page)

```

select * from sample_table;

/*-select all~sample_table-*/
select * from sample_table;

create or replace transient table any_other_table clone sample_table;

insert into any_other_table (
  select
    a.*
  from sample_table a
);

drop table if exists sample_table;
```

8.4.8 *tags_multi-line.sql*

Download

text

```
-- ..docs/snippets/script/tags_multi-line.sql

/*-
__name: I am a wrap
__description: This is an example of a wrap with the name explicitly declared.
-*/
select * from sample_table;

/*-
I am another wrap
__description: This is an example of a wrap with the name implicitly declared.
-*/
select * from sample_table;
```

8.4.9 *tags_single-line.sql*

Download

text

```
-- ..docs/snippets/script/tags_single-line.sql

/*-I am a wrap-*/
select * from sample_table;

/*-I am a wrap that isn't positioned correctly-*/

select * from sample_table;
```

8.5 Snowmobile

8.5.1 *connecting.py*

Download

text

```
""" Establish a basic connection.
../docs/snippets/connecting.py
"""
import snowmobile

sn = snowmobile.connect()

print(sn)             #> snowmobile.Snowmobile(creds='creds1')
print(sn.cfg)         #> snowmobile.Configuration('snowmobile.toml')
print(type(sn.con))   #> <class 'snowflake.connector.connection.SnowflakeConnection'>

sn2 = snowmobile.connect(creds="creds1")

sn.cfg.connection.current == sn2.cfg.connection.current  #> True
sn.current("schema") == sn2.sql.current("schema")     #> True
sn.current("session") == sn2.sql.current("session")   #> False

# -- complete example; should run 'as is' --
```

8.5.2 *connector_cursor_note.py*

Download

text

```
"""
Demonstrate instance exhaustion component of Connector.cursor.
../snippets/connector_cursor_note.py
"""
import snowmobile

sn = snowmobile.connect()

cur1 = sn.cursor.execute("select 1")
cur2 = sn.cursor.execute("select 2")

cursor = sn.cursor
cur11 = cursor.execute("select 1")
cur22 = cursor.execute("select 2")

id(cur1) == id(cur2)     #> False
id(cur11) == id(cur22)   #> True

# -- complete example; should run 'as is' --
```

8.5.3 *connector_delayed1.py*

Download

text

```
"""
Create a delayed snowmobile.Snowmobile object.
..docs/snippets/snowmobile/connector_delayed1.py
"""
import snowmobile

sn = snowmobile.connect(delay=True)

type(sn.con)      #> None
print(sn.alive)   #> False

_ = sn.query("select 1")

type(sn.con)      #> snowflake.connector.connection.SnowflakeConnection
print(sn.alive)   #> True

# -- complete example; should run 'as is' --
```

8.5.4 *connector_delayed2.py*

Download

text

```
"""
Demonstrate calling .connect() on existing Snowmobile instances.
..docs/snippets/snowmobile/connector_delayed2.py
"""
import snowmobile

# -- Delayed Connection --
sn_del = snowmobile.connect(delay=True)

print(type(sn_del.con))   #> None
sn_del.connect()
print(type(sn_del.con))   #> snowflake.connector.connection.SnowflakeConnection


# -- Live Connection --
sn_live = snowmobile.connect()

session1 = sn_live.sql.current('session')
sn_live.connect()
session2 = sn_live.sql.current('session')
print(session1 != session2)   #> True

# -- complete example; should run 'as is' --
```

8.5.5 *ensure_alive.py*

Download

text

```
"""
Demonstrate behavior of Connector's 'ensure_alive' parameter.
..docs/snippets/connector_ensure_alive.py
"""
import snowmobile

# --- SESSION #1 ---

# Explicitly providing default argument for clarity
sn = snowmobile.connect(ensure_alive=True)

print(sn.alive)  #> True
type(sn.con)     #> snowflake.connector.connection.SnowflakeConnection

# Storing 1st session ID
session1 = sn.current('session')

# Killing connection
sn.disconnect()

print(sn.alive)  #> False
type(sn.con)     #> NoneType

# --- SESSION #2 ---

# Calling any method requiring a connection
_ = sn.query("select 1")

# Storing 2nd session ID
session2 = sn.current('session')

# Verifying both session IDs are valid
print(type(session1))  #> str
print(type(session2))  #> str

# Verifying they're unique
print(session1 != session2)  #> True

# -- complete example; should run 'as is' --
```

8.5.6 *executing.py*

Download

text

```
"""Demonstrate primary methods for executing raw sql.
../docs/snippets/snowmobile/executing.py
"""
import snowmobile

sn = snowmobile.connect()

# -- sn.query() ---------------------------------------------------------------
df = sn.query("select 1")   #  == pd.read_sql()
type(df)                    #> pandas.core.frame.DataFrame

# -- pd.read_sql() --
import pandas as pd

df2 = pd.read_sql(sql="select 1", con=sn.con)

print(df2.equals(df))  #> True


# -- sn.ex() ------------------------------------------------------------------
cur = sn.ex("select 1")     #  == SnowflakeConnection.cursor().execute()
type(cur)                   #> snowflake.connector.cursor.SnowflakeCursor

# -- SnowflakeConnection.cursor().execute() --
cur2 = sn.con.cursor().execute("select 1")

print(cur.fetchone() == cur2.fetchone())  #> True


# -- sn.exd() -----------------------------------------------------------------
dcur = sn.exd("select 1")   #  == SnowflakeConnection.cursor(DictCursor).execute()
type(dcur)                  #> snowflake.connector.DictCursor

# -- SnowflakeConnection.cursor(DictCursor).execute() --
from snowflake.connector import DictCursor

dcur2 = sn.con.cursor(cursor_class=DictCursor).execute("select 1")

print(dcur.fetchone() == dcur2.fetchone())  #> True

# -- complete example; should run 'as is' --
```

8.5.7 *inspect_connector.py*

Download

text

```
"""
Instantiate a vanilla Snowmobile and inspect key attributes.
../docs/snippets/snowmobile/inspect_connector.py
"""
import snowmobile

sn = snowmobile.connect()

type(sn)          #> snowmobile.core.connection.Snowmobile

type(sn.cfg)      #> snowmobile.core.configuration.Configuration
str(sn.cfg)       #> snowmobile.Configuration('snowmobile.toml')

type(sn.con)      #> snowflake.connector.connection.SnowflakeConnection
type(sn.cursor)   #> snowflake.connector.cursor.SnowflakeCursor

# -- complete example; should run 'as is' --
```

8.5.8 *specifying_configuration.py*

Download

text

```
"""
Demonstrate specifying an alternate snowmobile.toml file *path*.
../docs/snippets/snowmobile/specifying_configuration.py
"""
from pathlib import Path

import snowmobile

path = Path.cwd() / 'snowmobile_v2.toml'  # any alternate file path

sn = snowmobile.connect(from_config=path)
```

8.5.9 *specifying_configuration2.py*

Download

text

```python
"""
Demonstrate specifying an alternate snowmobile.toml file *name*.
../docs/snippets/snowmobile/specifying_configuration2.py
"""
# -- SETUP ---------------------------------------------------------------------

import time
import shutil
import snowmobile

# Instantiate sn from snowmobile.toml; omit unnecessary connection
sn = snowmobile.connect(delay=True)

# Create alternate snowmobile.toml file called 'snowmobile2.toml'
path_cfg_orig = sn.cfg.location
path_cfg2 = path_cfg_orig.parent / 'snowmobile2.toml'
shutil.copy(path_cfg_orig, path_cfg2)


# -- EXAMPLE -------------------------------------------------------------------

def alt_sn(n: int) -> snowmobile.Snowmobile:
    """Instantiate sn from snowmobile2.toml and print time elapsed."""
    pre = time.time()
    sn = snowmobile.connect(
        config_file_nm='snowmobile2.toml',
        delay=True  # omit connection - not needed
    )
    print(f"n={n}, time-required: ~{int(time.time() - pre)} seconds")
    return sn


sn_alt1 = alt_sn(n=1)  #> n=1, time-required: ~6 seconds  -> locates file, caches path
sn_alt2 = alt_sn(n=2)  #> n=2, time-required: ~0 seconds  -> uses cache from sn_alt1
"""
Note:
    The time required for `sn_alt1` to locate 'snowmobile2.toml' is arbitrary and
    will vary based the file's location relative to the current working directory.
"""


# -- TEARDOWN ------------------------------------------------------------------
# Deleting 'snowmobile2.toml' from file system post-example

import os
os.remove(sn_alt1.cfg.location)
```

8.5.10 *verify_default_alias_change.py*

Download

text

```
"""
Verify `default-creds` has been changed to `creds2`.
../docs/snippets/snowmobile/verify_default_alias_change.py
"""
import snowmobile

sn = snowmobile.connect()

assert sn.cfg.connection.default_alias == 'creds2', (
    "Something's not right here; expected default_alias =='creds2'"
)
```

8.6 SQL

8.6.1 *sql_cross_schema.py*

Download

text

```
"""
Demonstrate prefixing object names with an alternative schema.
../docs/snippets/sql/sql_cross_schema.py
"""
import snowmobile

sn = snowmobile.connect()

# -- SETUP ------------------------------------------------------------------------
setup_sql = (
    """
    create or replace table sample_table as with
    sample_data as (
      select
        uniform(1, 10, random(1)) as rand_int
      from table(generator(rowcount => 3)) v
    )
      select
        row_number() over (order by a.rand_int) as col1
        ,(col1 * col1) as col2
      from sample_data a
    """
)

sn.ex(setup_sql)  # create 'sample_table'
```

(continues on next page)

(continued from previous page)

```

# -- EXAMPLE ------------------------------------------------------------------
import snowmobile
from snowflake.connector.errors import DatabaseError

try:
    # setup
    schema_nms = ['sample_schema1', 'sample_schema2']
    for schema in schema_nms:
        _ = snowmobile.connect().ex(f"create or replace schema {schema}")

    sn = snowmobile.connect()
    assert sn.current('schema').lower() not in schema_nms

    # ==================
    # - Start Example -
    # ==================

    # Clone some tables
    sn.clone(nm='sample_table', to='other_schema.sample_table')
    sn.drop(nm='other_schema.sample_table')
    sn.clone(  # other to current schema
        nm='other_schema.sample_table',
        to='sample_table',
    )

    # Query metadata
    print(sn.exists('sample_table'))                  #> True
    print(sn.exists('sample_schema.sample_table2'))   #> True
    print(sn.exists('gem7318.sample_table3'))                 #> True
    print(sn.current_schema())

    # sn.drop() works the same way
    for t in [
        'sample_table',
        'sample_schema.sample_table2',
        # 'sample_table3',
    ]:
        sn.drop(t)

    # ==================
    # - End Example -
    # ==================

except DatabaseError as e:
    raise e

finally:
    # teardown
    sn.drop(nm='sample_schema', obj='schema')
# snowmobile-include

sn = snowmobile.connect()
sn.current('schema').lower()


```

CHAPTER

NINE

ACKNOWLEDGEMENTS

9.1 API

appdirs

- The `AppDirs` class is used to determine the application data location across operating systems in `snowmobile.core.cache`.

pandas

- `pandas.sql.io.get_schema()` is used to generate generic DDL from a raw `DataFrame` within `snowmobile.Table`
- `pandas.read_sql()` is used to read the results of a query directly into a `DataFrame`

pydantic

- pydantic is used to define, parse, and validate the configuration model in *`snowmobile.core.cfg`*

sqlparse

- The `sqlparse.parsestream()` method is used for the **initial** parsing of a raw SQL file into individual st.

9.2 Documentation

Code Parsing

- **AutoAPI** is used to generate the *API reference documentation* from snowmobile's source code.

Docs Parsing

- The rest of the docs are built on top of the glorious work being done by the **The Executable Book Project**, specifically:
  - **MySt** & **MySt-NB**
  - **Sphinx-copybutton**
  - **Sphinx-togglebutton**
  - **Sphinx-tabs**

Theme

- **Material for Sphinx** is the base theme for this site

CHAPTER

TEN

CHANGELOG

10.1 v0.1.13

- Addition of `snowprocess` - background module, no user-facing changes

10.2 v0.1.12

- Removing `from_file` argument from `snowquery.query`
- Added manual commits to `snowloader` between commands ensure DDL execution is realized by the warehouse before data is attempted to load into table

10.3 v0.1.11

- `snowscripter`
  - Adding additional logic to strip comments from object such that script.run() only runs on executable sql

10.4 v0.1.10

10.5 v0.1.9

- Fixing issue with caching syncing up across classes

10.6 v0.1.8

10.7 v0.1.7

10.8 v0.1.6

- `snowscripter`
  - Addition of sql parsing logic for comment cleansing
- `snowcreds`
  - Additional caching logic

10.9 v0.1.5

- Docs addition only

10.10 v0.1.4

- Switching dynamic tags to include beta indicator

10.11 v0.1.3

- Quick patch of HTML tag causing explosion in the docs

10.12 v0.1.2

- `snowquery`
  - Change from `snowquery.Snowflake()` to `snowquery.Connector()` for semantic purposes / clarity of instantiation
- `snowscripter`
  - Addition of `snowscripter.Script` methods:
    * `.reload_source()`

    * `.get_statements()`
    * `.fetch()`
  - Addition of `snowscripter.Statement` methods
    * `.execute()` w/ keyword args `return_results`, `render`, and `describe`
    * `.render()`
    * `.raw()`

10.13 v0.1.1

- Simplifying `snowscripter.raw()`

10.14 v0.1.0

- Initial upload for Python 3.7 and 3.8

CHAPTER

ELEVEN

AUTHORS

- Grant E Murray mailto:gmurray203@gmail.com

CHAPTER

TWELVE

LICENSE

CHAPTER

THIRTEEN

OVERVIEW

13.1 *Connecting*

Connecting

`snowmobile.connect()` returns a *Snowmobile* whose purpose is to:

1. Locate, instantiate, and store *snowmobile.toml* as a `Configuration` object (`sn.cfg`)
2. Establish a connection to Snowflake and store the SnowflakeConnection (`sn.con`)
3. Serve as the primary entry point to the SnowflakeConnection and snowmobile APIs

The first time it's invoked, *Snowmobile* will find *snowmobile.toml* and cache its location; this step isn't repeated unless the file is moved, the cache is manually cleared, or a new version of snowmobile is installed.

With all arguments omitted, it will authenticate with the default credentials and connection arguments specified in *snowmobile.toml*.

Usage: Snowmobile snowmobile.core.connection

Establishing a connection from configured defaults is done with:

```
import snowmobile

sn = snowmobile.connect()
```

`sn` is a *Snowmobile* with the following attributes:

```
print(sn)             #> snowmobile.Snowmobile(creds='creds1')
print(sn.cfg)         #> snowmobile.Configuration('snowmobile.toml')
print(type(sn.con))   #> <class 'snowflake.connector.connection.SnowflakeConnection'>
```

Specific connection arguments are accessed by pre-configured alias:

```
sn2 = snowmobile.connect(creds='sandbox')

print(sn.cfg.connection.current != sn2.cfg.connection.current) #> True
```

sn

The variable `sn` represents a generic instance of *Snowmobile* roughly equivalent to that created with the snippet below; it's referred to as throughout the documentation, and applicable examples make use of it as a fixture without explicitly re-instantiating.

```
import snowmobile

sn = snowmobile.connect()
```

13.2 *Query Execution*

Query Execution

provides three convenience methods for executing raw SQL:

`query()` implements `pandas.read_sql()` for querying results into a `DataFrame`

`ex()` implements `SnowflakeConnection.cursor().execute()` for executing commands within a SnowflakeCursor

`exd()` implements `SnowflakeConnection.cursor(DictCursor).execute()` for executing commands within a DictCursor

Usage: Executing Raw SQL snowmobile.core.connection

Setup Assume a pre-existing **sample_table**:

COL1	COL2
1	1
2	4
4	9

Into a `DataFrame`:

```
sn.query('select * from sample_table')
```

```
   col1  col2
0     1     1
1     2     4
2     3     9
```

Into a SnowflakeCursor:

```
sn.ex('select * from sample_table').fetchall()
```

```
[(1, 1), (2, 4), (3, 9)]
```

Into a DictCursor:

```
sn.exd('select * from sample_table').fetchall()
```

```
[{'COL1': 1, 'COL2': 1}, {'COL1': 2, 'COL2': 4}, {'COL1': 3, 'COL2': 9}]
```

Or to get a single value:

```
sn.query('select count(*) from sample_table', as_scalar=True)
```

```
3
```

13.3 *Information API*

Information API

inherits everything from a ***SQL*** class that generates and executes raw SQL from inputs; its purpose is to provide a bare bones Python API to query metadata and execute administrative commands against Snowflake.

Usage: SQL snowmobile.core.sql

Check existence:

```
sn.exists('sample_table')  #> True
```

Select records:

```
sn.select('sample_table', n=1)
```

```
   col1  col2
0     1     1
```

Query metadata:

```
sn.count('sample_table')                  #> 3
sn.count('sample_table', dst_of='col1')   #> 3
sn.columns('sample_table')                #> ['COL1', 'COL2']
```

Verify dimensionality:

```
sn.is_distinct('sample_table', field='col1')  #> True
```

Submit basic administrative commands:

```
sn.clone(nm='sample_table', to='sample_table2')
```

Fetch DDL:

```
print(sn.ddl('sample_table'))
```

```
create or replace TABLE SAMPLE_TABLE (
        COL1 FLOAT,
        COL2 FLOAT
);
```

Drop objects:

```
for t in ['sample_table', 'sample_table2']:
    sn.drop(t, obj='table')

sn.exists('sample_table')    #> False
sn.exists('sample_table2')   #> False
```

13.4 *Loading Data*

Loading Data

-

`Table` is a loading solution that at minimum accepts a `df` (`DataFrame`) and a `table` name (`str`).

In the same way that ***Snowmobile*** handles its keyword arguments, `Table` will adhere to any arguments explicitly provided and defer to the values configured in *snowmobile.toml* otherwise.

More Info

The behavior outlined below reflects those within the default snowmobile.toml file, meaning that `t1` will:

1. Check if *sample_table* exists in the schema associated with `sn.con`
2. If *sample_table* **does** exist, it will validate `df` against *sample_table* and throw an error if their dimensions are not identical
3. If *sample_table* does **not** exist (as is the case here), it will generate DDL from `df` and execute it as part of the loading process

Usage: snowmobile.Table snowmobile.core.table

Setup

Assume *sample_table* does not yet exist and `df` was created with:

```
import pandas as pd
import numpy as np

df = pd.DataFrame(
    data = {'COL1': [1, 2, 3], 'COL2': [1, 4, 9]}
)
print(df.shape)  #> (3, 2)
```

COL1	COL2
1	1
2	4
3	9

Given ⚙, instantiating the following ***Table***, `t1`, with *as_is=True* will:

1. Generate and execute DDL to create *sample_table*
2. Load `df` into *sample_table* via the Bulk Loading from a Local File System standard

```
import snowmobile

t1 = snowmobile.Table(
    df=df,
    table='sample_table',
    as_is=True,
)
```

In absence of providing an existing to *`snowmobile.Table()`*, an instance was created and stored as a public attribute; the create and load for *sample_table* can be verified with either of:

```
print(t1.sn.exists('sample_table'))   #> True
print(t1.loaded)                      #> True
```

When compatability between the *df* and the *table* is unknown, *as_is=True* can be omitted and the ***Table*** that's returned inspected further prior to continuing with the loading process:

```
df2 = pd.concat([df, df], axis=1)
print(list(df2.columns))               #> ['COL1', 'COL2', 'COL1', 'COL2']
print(t1.sn.columns('sample_table'))   #> ['COL1', 'COL2']

t2 = snowmobile.Table(
    df=df2,
    table='sample_table',
)
```

Primary compatability checks are centered around things like:

```
print(t2.exists)       #> True
print(t2.cols_match)   #> False
```

With *snowmobile.toml* defaults, calling *`Table.load()`* on `t2` will throw an error:

```
from snowmobile.core.errors import ColumnMismatchError

try:
    t2.load()
except ColumnMismatchError as e:
    print(e)
"""
>>>
ColumnMismatchError: `SAMPLE_TABLE` columns do not equal those in the local DataFrame␣
↪and
if_exists='append' was specified.
```

(continues on next page)

(continued from previous page)

```
Either provide if_exists='replace' to overwrite the existing table or see `table.col_
↪diff()`
to inspect the mismatched columns.
"""
```

Table.col_diff() returns a dictionary of tuples containing the table and DataFrame columns by index position; providing *mismatch=True* limits the results to only those responsible for the `ColumnMismatchError`:

```
import json

print(json.dumps(t2.col_diff(mismatched=True), indent=4))
"""
>>>
{
  "3": [
    null,
    "col1_1"
  ],
  "4": [
    null,
    "col2_1"
  ]
}
"""
```

Keyword arguments take precedent over configurations in *snowmobile.toml*, so `df2` can still be loaded with:

```
t2.load(if_exists='replace')

print(t2.loaded)                          #> True
print(t2.sn.columns('sample_table'))      #> ['COL1', 'COL2', 'COL1_1', 'COL2_1']
```

Note: With default behavior, the duplicate column names in `df2` were automatically renamed by `t2` before loading into *sample_table*:

```
print(list(df2.columns))     #> ['COL1', 'COL2', 'COL1', 'COL2']
print(list(t2.df.columns))   #> ['COL1', 'COL2', 'COL1_1', 'COL2_1']
```

13.5 *Working with SQL Scripts*

Working with SQL Scripts

-

snowmobile.Script accepts a full `path` to a sql file and parses its contents based on patterns specified in *snowmobile.toml*.

At a minimum, the file is split into individual st, each of which is checked for decorated information in the form of a string directly preceding it wrapped in an opening (`/*-`) and closing (`-*/`) pattern, the simplest form of which is a single-line string that can be used as an accessor to the statement it precedes.

When no information is provided, *Script* generates a generic name for the statement based on the literal first SQL keyword it contains and its index position.

More Info

Line **27** within *sample_table.sql* represents the minimum markup required to associate a name with an individual statement; consistency in tag structure has obvious benefits, but this is a freeform string that can be anything.

Line **19** is an example of a special tag; the leading `qa-empty` tells ***Script*** to run assertion that its results are null (0 records) before continuing execution of the script.

The tags for statements beginning on lines **1**, **6**, and **17** were generated by ***Script*** based their contents and relative positions within the script.

Usage: snowmobile.Script snowmobile.core.script

Setup

`path` is a full path (`pathlib.Path` or `str`) to a file, *sample_table.sql*, containing 5 standard sql st:

```
1  create or replace table sample_table (
2          col1 number(18,0),
3          col2 number(18,0)
4  );
5
6  insert into sample_table with
7  sample_data as (
8    select
9      uniform(1, 10, random(1)) as rand_int
10   from table(generator(rowcount => 3)) v
11 )
12   select
13     row_number() over (order by a.rand_int) as col1
14     ,(col1 * col1) as col2
15   from sample_data a;
16
17 select * from sample_table;
18
19 /*-qa-empty~verify 'sample_table' is distinct on 'col1'-*/
20 select
21   a.col1
```

(continues on next page)

(continued from previous page)

```
  ,count(*)
from sample_table a
group by 1
having count(*) <> 1;

/*-insert into~any_other_table-*/
insert into any_other_table (
  select
    a.*
    ,tmstmp.tmstmp as insert_tmstmp
  from sample_table a
  cross join (select current_timestamp() as tmstmp)tmstmp
);
```

sample_table.sql

Given a `path` to *sample_table.sql*, a *Script* can be created with:

```
import snowmobile

script = snowmobile.Script(path=path, silence=True)

print(script)         #> snowmobile.Script('sample_table.sql')
print(script.depth)   #> 5
script.dtl()
```

```
sample_table.sql
================
1: Statement('create table~s1')
2: Statement('insert into~s2')
3: Statement('select data~s3')
4: Statement('qa-empty~verify 'sample_table' is distinct on 'col1'')
5: Statement('insert into~any_other_table')
```

Statements can be accessed by their index position or name (*nm*):

```
script(5)                              #> Statement('insert into~any_other_table')
script(-1)                             #> Statement('insert into~any_other_table')
script('insert into~any_other_table') #> Statement('insert into~any_other_table')
```

Each *Statement* has its own set of attributes:

```
s3 = script(3)  # store 3rd statement

print(s3.index)   #> 3
print(s3.sql)     #> select * from sample_table
print(s3.kw)      #> select
print(s3.desc)    #> sample_table
print(s3.nm)      #> select~sample_table
```

Based on statement attributes, `script` can be filtered and used within that context:

```
with script.filter(
    excl_desc=['.*any_other_table'],  # throwing out s5; pattern is regex not glob
    excl_kw=['select'],               # throwing out s3
```

(continues on next page)

(continued from previous page)

```
) as s:
    s.run()
```

```
sample_table.sql
================
<1 of 3> create table~s1 (1s)....................................... <completed>
<2 of 3> insert into~s2 (0s)........................................ <completed>
<3 of 3> qa-empty~verify 'sample_table' is distinct on 'col1' (0s).... <passed>
```

Spans of statements are directly executable by index boundaries:

```
script.run((1, 3))
```

```
sample_table.sql
================
<1 of 6> create table~s1 (0s)............................... <completed>
<2 of 6> insert into~s2 (0s)................................ <completed>
<3 of 6> select data~s3 (0s)................................ <completed>
```

And their results accessible retroactively:

```
script(3).results.head()
```

```
   col1  col2
0     1     1
1     2     4
2     3     9
```

PYTHON MODULE INDEX

S

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W