Pallas

Miloslav Pojman

Jan 06, 2022

CONTENTS

Pallas makes querying AWS Athena easy.

It is especially valuable for analyses in Jupyter Notebook, but it is designed to be generic and usable in any application.

Main features:

- Friendly interface to AWS Athena.
- Caching – local and remote cache for reproducible results.
- Performance – Large results are downloaded directly from S3.
- Pandas integration - Conversion to DataFrame with appropriate dtypes.
- Optional white space normalization for better caching.
- Kills queries on KeyboardInterrupt.

```
import pallas
athena = pallas.environ_setup()
df = athena.execute("SELECT 'Hello world!'").to_df()
```

Pallas is hosted at GitHub and it can be installed from PyPI.

This documentation is available online at Read the Docs.

CHAPTER

ONE

TABLE OF CONTENTS

1.1 Installation

Pallas requires Python 3.7 or newer. It can be installed using pip:

```
pip install pallas
```

When Pandas are installed, query results can be converted to `pandas.DataFrame`.

```
pip install pallas[pandas]
```

1.2 Tutorial

1.2.1 AWS credentials

Pallas uses boto3 internally, so it reads AWS credentials from the standard locations:

- Shared credential file (`~/.aws/credentials`)
- Environment variables (`AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY`)
- Instance metadata service when run on an Amazon EC2 instance

The `~/.aws/credentials` file can be generated using the AWS CLI.

```
aws configure
```

We recommend to use the AWS CLI to check the configuration. If the AWS CLI is able to authenticate then Pallas should work too.

```
aws sts get-caller-identity
aws athena list-databases --catalog-name AwsDataCatalog
```

1.2.2 Initialization

An *Athena* client can be obtained using the *setup()* function. All arguments are optional.

```
import pallas
athena = pallas.setup(
    # AWS region, read from ~/.aws/config if not specified.
    region=None,
    # Athena (AWS Glue) database.
    database=None,
    # Athena workgroup. Will use default workgroup if omitted.
    workgroup=None,
    # Athena output location, will use workgroup default location if omitted.
    output_location="s3://...",
    # Optional query execution cache.
    cache_remote="s3://...",
    # Optional query result cache.
    cache_local="~/Notebooks/.cache/",
    # Whether to return failed queries from cache. Defaults to False.
    cache_failed=False,
    # Normalize white whitespace for better caching. Enabled by default.
    normalize=True,
    # Kill queries on KeybordInterrupt. Enabled by default.
    kill_on_interrupt=True
)
```

To avoid hardcoded configuration values, the *environ_setup()* function can initialize *Athena* from environment variables, corresponding to arguments in the previous example:

```
export PALLAS_REGION=
export PALLAS_DATABASE=
export PALLAS_WORKGROUP=
export PALLAS_OUTPUT_LOCATION=
export PALLAS_NORMALIZE=true
export PALLAS_KILL_ON_INTERRUPT=true
export PALLAS_CACHE_REMOTE=$PALLAS_OUTPUT_LOCATION
export PALLAS_CACHE_LOCAL=~/Notebooks/.cache/
export PALLAS_CACHE_FAILED=false
```

```
athena = pallas.environ_setup()
```

Pallas uses Python standard logging. You can use *configure_logging()* instead of `logging.basicConfig()` to enable logging for Pallas only. At the DEBUG level, Pallas emits logs with query status including an estimated price:

```
pallas.configure_logging(level="DEBUG")
```

1.2.3 Executing queries

Use the *Athena.execute()* method to execute queries:

```
sql = "SELECT %s id, %s name, %s value"
results = athena.execute(sql, (1, "foo", 3.14))
```

Pallas also support non-blocking query execution:

```
query = athena.submit(sql)  # Submit a query and return
query.join()  # Wait for query completion.
results = query.get_results()  # Retrieve results. Joins the query internally.
```

The result objects provides a list-like interface and can be converted to a Pandas DataFrame:

```
df = results.to_df()
```

1.2.4 Caching

AWS Athena stores query results in S3 and does not delete them, so all past results are cached implicitly. To retrieve results of a past query, an ID of the query execution is needed.

Pallas can cache in two modes - remote and local:

- In the remote mode, Pallas stores IDs of query executions. Using that, it can download previous results from S3 when they are available.
- In the local mode, it copies query results. Thanks to that, locally cached queries can be executed without an internet connection.

Note: Pallas is designed to promote reproducible analyses and data pipelines:

- Using the local caching, it is possible to regularly restart Jupyter notebooks without waiting for or paying for additional Athena queries.
- Thanks to the remote caching, results can be reproduced at a different machine by a different person.

Reproducible queries should be deterministic. For example, if you query data that are ingested regularly, you should always filter on the date column.

Pallas assumes that your queries are deterministic and does not invalidate its cache.

Caching configuration can be passed to *setup()* or *environ_setup()*, as shown in the *Initialization* section.

After the initialization, caching can be customized later using the *Athena.cache* property:

```
athena.cache.enabled = True  # Default
athena.cache.read = True  # Can be set to False to write but not read the cache
athena.cache.write = True  # Can be set to False to read but not write the cache
athena.cache.local = "~/Notebooks/.cache/"
athena.cache.remote = "s3://..."
athena.cache.failed = True
```

Alternatively, the *Athena.using()* method can override a configuration for selected queries only:

```
athena.using(cache_enabled=False).execute(...)
```

Only SELECT queries are cached.

1.3 API

This page describes the public API of the Pallas library.

All public functions and classes are imported to the top level `pallas` module. Imports from internals of the package are not recommended and can break in future.

1.3.1 Assembly

To construct an *Athena* client, use *setup()* or *environ_setup()* functions.

setup(*, *region=None*, *database=None*, *workgroup=None*, *output_location=None*, *cache_local=None*, *cache_remote=None*, *cache_failed=False*, *normalize=True*, *kill_on_interrupt=True*)

Setup an *Athena* client.

All configuration options can be given to this method, but many of them can be overridden after the client is constructed.

Parameters

- **region** (*str | None*) – an AWS region. By default, region from AWS config (`~/.aws/config`) is used.
- **database** (*str | None*) – a name of Athena database. Can be overridden in SQL.
- **workgroup** (*str | None*) – a name of Athena workgroup. Workgroup can set resource limits or override output location. Defaults to the Athena default workgroup.
- **output_location** (*str | None*) – an output location at S3 for query results. Optional if an output location is specified for the *workgroup*.
- **cache_local** (*str | None*) – an URI of a local cache. Both results and query execution IDs are stored in the local cache.
- **cache_remote** (*str | None*) – an URI of a remote cache. Query execution IDs without results are stored in the remote cache.
- **cache_failed** (*bool*) – whether to return failed queries found in cache.
- **normalize** (*bool*) – whether to normalize queries before execution.
- **kill_on_interrupt** (*bool*) – whether to kill queries on KeyboardInterrupt.

Returns a new instance of Athena client

Return type *Athena*

environ_setup(*environ=None*, *, *prefix='PALLAS'*)

Setup an *Athena* client from environment variables.

Reads the following environment variables:

```
export PALLAS_REGION=
export PALLAS_DATABASE=
export PALLAS_WORKGROUP=
export PALLAS_OUTPUT_LOCATION=
export PALLAS_NORMALIZE=true
export PALLAS_KILL_ON_INTERRUPT=true
export PALLAS_CACHE_REMOTE=$PALLAS_OUTPUT_LOCATION
export PALLAS_CACHE_LOCAL=~/Notebooks/.cache/
```

Configuration from the environment variables can be overridden after the client is constructed.

Parameters

- **environ** (*Mapping[str, str] | None*) – A mapping object representing the string environment. Defaults to `os.environ`.
- **prefix** (*str*) – A prefix of environment variables

Returns a new instance of Athena client

Return type *Athena*

configure_logging(*, *level=20*, *stream=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='UTF-8'>*, ***kwargs*)

Do basic configuration for the logging system.

Calls `logging.basicConfig()` internally, but:

- Sets *level* to the "pallas" logger only
- Log level default to "INFO:
- *stream* defaults to `sys.stdout` instead of `sys.stderr`

Can be safely called no matter whether logging was already configured:

- If logging was already configured, this function just sets a level for the "pallas" logger.
- If logging was not configured yet, it enabled logging to stdout.

Parameters

- **level** (*int | str*) – Set the "pallas" logger to the specified level.
- **stream** (*TextIO*) – Use the specified stream to initialize the StreamHandler.
- **kwargs** – passed to `logging.basicConfig()`

Return type None

1.3.2 Client

The *Athena* class is a facade to all functionality offered by the library.

In the most common scenario, you may need only its *execute()* method. If you need to submit queries in a non-blocking fashion, you can use the *submit()* method, which returns a *Query* instance. The same class is also returned by *get_query()* method, which can be useful if you want to get back to queries executed in the past.

class Athena(*proxy*)

Athena client.

Provides methods to execute SQL queries in AWS Athena, with an optional caching and other helpers.

Can be used as a blocking or a non-blocking client.

Use *setup()* or *environ_setup()* to construct this class without touching Pallas internals.

Parameters **proxy** – an internal proxy to execute queries

static quote(*value*)

Quote a scalar value for an SQL expression.

Parametrized queries should be preferred to explicit quoting.

Following Python types can be quoted to an SQL expressions:

- `None` – SQL `NULL`

- `str`
- `int`, including subclasses of numbers.Integral
- `float`, including subclasses or numbers.Real
- `Decimal` – SQL `DECIMAL`
- `datetime.date` – SQL `DATE`
- `datetime.datetime` – SQL `TIMESTAMP`
- `bytes` – SQL `VARBINARY`

Parameters **value** (*Union[None, str, float, numbers.Real, decimal.Decimal, bytes, datetime.date]*) – Python value

Returns an SQL expression

Return type str

database: str | None = None
Name of Athena database to be be queried.

Can be overridden in SQL.

workgroup: str | None = None
Name of Athena workgroup.

Workgroup can set resource limits or override output location. When None, defaults to the Athena default workgroup.

output_location: str | None = None
URI of output location on S3.

Optional if an output location is specified for *workgroup*.

normalize: bool = True
Whether to normalize queries before execution.

kill_on_interrupt: bool = True
Whether to kill queries on KeyboardInterrupt

property cache
Cache implementation.

It is possible to update properties of the *cache* attribute to reconfigure caching in place.

Alternatively, the *using()* method can apply a new configuration without affecting an existing instance.

Return type *AthenaCache*

using(*, *database=None*, *workgroup=None*, *output_location=None*, *normalize=None*, *kill_on_interrupt=None*, *cache_enabled=None*, *cache_read=None*, *cache_write=None*, *cache_failed=None*)
Crate a new instance with an updated configuration.

This method can be useful if you need to override a configuration for one query, but you do not want to affect future queries.

Parameters

- **database** (*str | None*) – name of Athena database to be be queried.
- **workgroup** (*str | None*) – name of Athena workgroup.
- **output_location** (*str | None*) – URI of output location on S3.

- **normalize** (*bool | None*) – whether to normalize queries before execution.
- **kill_on_interrupt** (*bool | None*) – whether to kill queries on KeyboardInterrupt
- **cache_enabled** (*bool | None*) – whether a cache should be used.
- **cache_read** (*bool | None*) – whether a cache should be read.
- **cache_write** (*bool | None*) – whether a cache should be written.
- **cache_failed** (*bool | None*) – whether to return failed queries found in cache.

Returns an updated copy of this client

Return type *Athena*

execute(*operation*, *parameters=None*)
Execute a query and return results.

This is a blocking method that waits until the query finishes.

Cached results or results from an existing query can be returned, if the caching was configured. Only SELECT queries are cached.

Raises *AthenaQueryError* if the query fails.

Parameters

- **operation** (*str*) – an SQL query to be executed Can contain `%s` or `%(key)s` placeholders for substitution by *parameters*.
- **parameters** (*Union[None, Tuple[SQL_SCALAR, ..], Mapping[str, SQL_SCALAR]]*) – parameters to substitute in *operation*. All substitute parameters are quoted appropriately. See the *quote()* method for a supported parameter types.

Returns query results

Return type *pallas.results.QueryResults*

submit(*operation*, *parameters=None*)
Submit a query and return.

This is a non-blocking method that starts a query and returns. Returns a *Query* instance for monitoring query execution and downloading results later.

An existing query can be returned, if the caching was configured. Only SELECT queries are cached.

Parameters

- **operation** (*str*) – an SQL query to be executed Can contain `%s` or `%(key)s` placeholders for substitution by *parameters*.
- **parameters** (*Union[None, Tuple[SQL_SCALAR, ..], Mapping[str, SQL_SCALAR]]*) – parameters to substitute in *operation*. All substitute parameters are quoted appropriately. See the *quote()* method for a supported parameter types.

Returns a query instance

Return type *pallas.client.Query*

get_query(*execution_id*)
Get a previously submitted query execution.

This method can be used to retrieve a query executed in the past. Because Athena stores results in S3 and does not delete them by default, it is possible to download results until they are manually deleted.

Parameters **execution_id** (*str*) – an Athena query execution ID.

Returns a query instance

Return type *pallas.client.Query*

class Query(*execution_id*, *, *proxy*, *cache*)

Athena query

Provides access to one query execution. It can be used to monitor status of the query results or retrieving results when the execution finishes.

Instances of this class are returned by *Athena.submit()* and *Athena.get_query()* methods. You should not need to create this class directly.

Parameters

- **execution_id** – Athena query execution ID.
- **proxy** – an internal proxy to execute queries
- **cache** – a cache instance

backoff: Iterable[int] = <pallas.utils.Fibonacci object>

Delays in seconds between for checking query status.

kill_on_interrupt: bool = False

Whether to kill this query on KeyboardInterrupt

Initially set to *Athena.kill_on_interrupt*.

property execution_id

Athena query execution ID.

This ID can be used to retrieve this query later using the *Athena.get_query()* method.

get_info()

Retrieve information about this query execution.

Returns a status of this query with other information.

Return type *pallas.info.QueryInfo*

get_results()

Download results of this query execution.

Cached results can be returned, if the caching was configured. Only SELECT queries are cached.

Waits until this query execution finishes and downloads results. Raises *AthenaQueryError* if the query failed.

Return type *pallas.results.QueryResults*

kill()

Kill this query execution.

This is a non-blocking operation. It does not wait until the query is killed.

Return type None

join()

Wait until this query execution finishes.

Raises *AthenaQueryError* if the query failed.

Return type None

1.3.3 Query information

Information about query execution are returned as *QueryInfo* instances. If you call *Query.get_info()* multiple times, it can return different information as the query execution proceeds.

class QueryInfo(*data*)
Information about query execution.

Instances are returned by the *Query.get_info()* method.

Parameters **data** – data returned by Athena GetQueryExecution API method.

__str__()
Return summary info about the query execution.

This is included in logs generated by the Athena client.

Return type str

property execution_id
ID od the query execution.

property sql
SQL query executed.

property output_location
URI of output location on S3 for the query

property database
Name of database.

property finished
Whether the query execution finished.

property succeeded
Whether the query execution finished successfully.

property state
State of the query execution.

property state_reason
Reason of the state of the query execution.

property scanned_bytes
Data scanned by Athena.

property execution_time
Time spent by Athena.

check()
Raises *AthenaQueryError* (or its subclass) if the query failed.

Does not raise if the query is still running.

Return type None

1.3.4 Query results

Results of query executions are encapsulated by the *QueryResults* class.

class QueryResults(*column_names*, *column_types*, *data*)
Collection of Athena query results.

Implements a list-like interface for accessing individual records. Alternatively, can be converted to `pandas.DataFrame` using the *to_df()* method.

__getitem__(*index*)
Return one result or slice of results.

Records are returned as mappings from column names to values.

Parameters **index** (*int | slice*) –

Return type QueryRecord | Sequence[QueryRecord]

__len__()
Return count of this results.

Return type int

classmethod load(*stream*)
Deserialize results from a text stream.

Parameters **stream** (*TextIO*) –

Return type *pallas.results.QueryResults*

save(*stream*)
Serialize results to a text stream.

Parameters **stream** (*TextIO*) –

Return type None

property column_names
List of column names.

property column_types
List of column types.

to_df(*dtypes=None*)
Convert this results to `pandas.DataFrame`.

Parameters **dtypes** (*Mapping[str, object] | None*) –

Return type pd.DataFrame

1.3.5 Caching

class AthenaCache
Caches queries and its results.

Athena always stores results in S3, so it is possible to retrieve past results without manually copying data.

This class can hold a reference to two instances of cache storage:

- local, which caches both query execution IDs and query results
- remote, which cache query execution IDs only.

It is possible to configure one the backends, both of them, or none of them.

Queries cached in the local storage can be executed without an internet connection. Queries cached in the remote storage are not executed twice, but results have to be downloaded from AWS.

In theory, it is possible to use remote backend for the local cache (or vice versa), but we assume that the local cache is actually stored locally

Instance of this class is returned by the *Athena.cache* property. It can be updated to reconfigure the caching.

enabled: bool = True
Can be set to False to disable caching completely.

Can be updated to enable or disable the caching.

read: bool = True
Can be set to False to disable reading the cache.

Can be updated to reconfigure the caching.

write: bool = True
Can be set to False to disable writing the cache.

Can be updated to reconfigure the caching.

failed: bool = False
Whether to return failed queries found in cache.

When this is false, failed queries found in cache are ignored.

property local
URI of storage for local cache.

Can be updated to reconfigure the caching.

property remote
URI of storage for remote cache.

Can be updated to reconfigure the caching.

load_execution_id(*database*, *sql*)
Retrieve cached query execution ID for the given SQL.

Looks into both the local and the remote storage.

Parameters

- **database** (*str | None*) –
- **sql** (*str*) –

Return type str | None

save_execution_id(*database*, *sql*, *execution_id*)
Store cached query execution ID for the given SQL.

Updates both the local and the remote storage.

Parameters

- **database** (*str | None*) –
- **sql** (*str*) –
- **execution_id** (*str*) –

Return type None

has_results(*execution_id*)
Checks whether results are cached for the given execution ID.

Looks into the local storage only.

Parameters **execution_id** (*str*) –

Return type bool

load_results(*execution_id*)
Retrieve cached results for the given execution ID.

Looks into the local storage only.

Parameters **execution_id** (*str*) –

Return type QueryResults | None

save_results(*execution_id*, *results*)
Store cached results for the given SQL.

Updates the local storage only.

Parameters

- **execution_id** (*str*) –
- **results** (pallas.results.QueryResults) –

Return type None

1.3.6 Exceptions

Pallas can raise *AthenaQueryError* when a query fails. For transport errors (typically connectivity problems or authorization failures), `boto3` exceptions bubble unmodified.

class AthenaQueryError(*execution_id*, *state*, *state_reason*)
Athena query failed.

state: str
State of the query execution (FAILED or CANCELLED)

state_reason: str | None
Reason of the state of the query execution.

__str__()
Report query state with its reason.

Return type str

class DatabaseNotFoundError(*execution_id*, *state*, *state_reason*)
Bases: *pallas.exceptions.AthenaQueryError*

Athena database does not exist.

Pallas maps string errors returned by Athena to exception classes.

class TableNotFoundError(*execution_id*, *state*, *state_reason*)
Bases: *pallas.exceptions.AthenaQueryError*

Athena table does not exist.

Pallas maps string errors returned by Athena to exception classes.

1.4 Development

1.4.1 Installation

Pallas can be installed with development dependencies using pip:

```
pip install -e .[dev]
```

1.4.2 Configuration

For integration test to run, access to AWS resources has to be configured.

```
export TEST_PALLAS_REGION=            # AWS region, can be also specified in ~/.aws/
↪config
export TEST_PALLAS_DATABASE=          # Name of Athena database
export TEST_PALLAS_WORKGROUP=         # Optional
export TEST_PALLAS_OUTPUT_LOCATION=   # s3:// URI
```

If the above environment variables are not defined, integration tests will be skipped.

1.4.3 Tools

- Code is checked with flake8 and Mypy.
- Tests are run using pytest.
- Code is formatted using Black and isort.
- Documentation is built using Sphinx.

Tox can run the above tools:

```
tox -e format
tox --parallel
```

1.5 Alternatives

PyAthena and AWS Data Wrangler are good alternatives to Pallas. They are more widespread and presumably more mature than Pallas.

1.5.1 Intro

The main benefit of Pallas is the powerful caching designed for workflows in Jypyter Notebook. Thanks to the local cache, it is possible to restart notebooks often without waiting for data. The cache in S3 allows to reproduce results from teammates without incurring additional costs.

Pallas offers small but useful helpers. Query normalization allows to write nicer (indented) code without impact on caching. Estimated price in logs and kill on KeyboardInterrupt can help you to control costs.

Pallas has an opinionated API, which does not implement Python DB API nor copies boto3.

- Unlike Python DB API, Pallas interface embraces asynchronous execution. It allows to retrieve past queries by their ID and download old results.

- Pallas does not follow procedural style of boto3. A client object holds all necessary configuration, and query objects encapsulates everything related to query executions.

1.5.2 PyAthena

PyAthena is a Python DB API 2.0 (PEP 249) compliant client for Amazon Athena. It is integrated with Pandas and SQLAlchemy.

Pallas vs PyAthena

- PyAthena is older and more popular.
- Pallas does not offer Python DB API or SQLAlchemy integration.
- PyAthena uses a distinct cursor type for execution in a background thread. Pallas can submit a query without waiting for results and offers a Query class for monitoring or joining the query.
- PyAthena can list last N queries when looking for cached results. Pallas can cache queries locally and to S3, so the cache is unlimited and can work offline.
- PyAthena downloads results directly from S3 only if PandasCursor is used.
- PyAthena uses Pandas for reading CSV files. Pallas implements own CSV parser with explicit mapping from Athena types to Pandas types.
- Pallas does not have helpers for creating new tables.

1.5.3 AWS Data Wrangler

AWS Data Wrangler integrates Pandas with many AWS services, including Athena.

Interface of its Athena client is very similar to the boto3 API. Function names copy function methods from boto3, but invocation is simplified thanks to flattened arguments.

AWS Data Wrangler uses an interesting trick to obtain results in Parquet format. Its CTAS approach rewrites `SELECT` queries to `CREATE TABLE` statements, and then reads Parquet output from S3. Advantages of the CTAS approach are performance and handling of complex types that cannot be read from CSV.

Pallas vs AWS Data Wrangler

- AWS Data Wrangler is a part of AWS Labs and is managed by AWS Professional Services.
- Pallas does not offer the CTAS approach, but it downloads CSV files from S3. The main performance improvement comes from bypassing Athena API. CSV parsing can be slower than reading Parquet, but this difference should be negligible compared to the time spent downloading data.
- AWS Data Wrangler lists last N queries when looking for cached results. Pallas can cache queries locally and to S3, so the cache is unlimited and can work offline.
- AWS Data Wrangler uses on pyarrow for reading Parquet files and Pandas for reading CSV files. Pallas implements own CSV parser with explicit mapping from Athena types to Pandas types.
- Pallas does not mimic boto3 API, it provides object interface instead.
- Pallas misses helpers to call `MSCK REPAIR TABLE` or create an S3 bucket for AWS results.

1.5.4 boto3

boto3 is the official AWS SDK for Python. Pallas uses boto3 internally.

Querying Athena using boto3 directly is complicated and requires a lot of boilerplate code.

1.6 License

1.7 Changelog

1.7.1 v0.11.dev

Nothing yet.

1.7.2 v0.10 (2022-01-06)

- Failed or cancelled queries found in cached are ignored. Export `PALLAS_CACHE_FAILED=true` to use failed queries from the cache.
- A small optimization to avoid duplicate check of local cache.
- Use PEP 585 type annotations.
- Test with Python 3.9 and Python 3.10
- Test environment variables consistent with runtime environment variables.
- Refactor and cleanup of tests.

1.7.3 v0.9 (2021-03-03)

- Better logging. Log summary at INFO level and details at DEBUG level. Add a helper for logging configuration.
- Include QueryExecutionId in exception messages.
- Fix conversion because Athena sometimes returns "real" instead of "float".

1.7.4 v0.8 (2020-10-06)

- Remove deprecated ignore_cache parameter.
- Fix query execution ID not cached locally when cached remotely.

1.7.5 v0.7 (2020-08-31)

- Export new exceptions introduced v0.6 to the top level module.

1.7.6 v0.6 (2020-08-31)

- Raise *AthenaQueryError* subclasses when a database or a table is not found.
- Add more configuration options to the *Athena.using()* method.

1.7.7 v0.5 (2020-08-19)

- Do not substitute parameters (require quoted percent signs) when no parameters are given.

1.7.8 v0.4 (2020-08-18)

- Add support for parametrized queries.
- More options for cache configuration.
- Allow to override configuration of the Athena class after it is initialized.
- Refactored implementation from layered decorators to one class using specialized helpers.
- New documentation.
- All public (documented) functions and classes are available the top-level module.

1.7.9 v0.3 (2020-06-18)

- Athena and Query classes available from the top-level module (useful for type hints).
- AthenaQueryError from the top-level module.
- Fix: SELECT queries cached only when uppercase.
- Fix: Queries not killed on KeyboardInterrupt.

1.7.10 v0.2 (2020-06-02)

- Cache SELECT statements only (starting with SELECT or WITH).
- Preserve empty lines in the middle of normalized queries.

1.7.11 v0.1 (2020-03-24)

- Initial release.

1.8 Indices and tables

- genindex
- modindex
- search

PYTHON MODULE INDEX

p

INDEX

Symbols

A

B

C

D

E

F

G

H

J

K

L

M

N

O

P

Q

R

S

T

U

W