bis_code_helpers package

Introduction

The bis_code_helpers package is a set of ease-of-use functions that allows one to build standardised Python software at Wits BIS.

Installation

To install this package, run the following within the directory containing *setup.py*:

```
pip install .
```

This will install ensure the package's dependencies are installed and then install the package in the currently active python environment.

Usage

The package includes a bunch of functions that may be useful and don't necessarily follow one theme.

A major component of this package is database interaction. All functions that interact with a database take a SQLAlchemy *engine* argument, creates a temporary connection for that interaction using the engine, performs the interaction and then closes the connection.

Here is an example:

1. Import the package:

```
import bis_code_helpers
```

2. Create an engine:

```
USER: str = "Kyle"
PASS: str = "1234"
DB: str = "bi.example.org:1234/SERVICE"

engine = bis_code_helpers.create_engine(USER, PASS, DB)
```

3. Get the list of column names from a table on the DB linked to the engine:

```
TABLE_NAME: str = "TEST_TABLE"

column_names: list = bis_code_helpers.get_db_table_column_names(TABLE_NAME, engine)
```

Mock Logging

bis_code_helpers.set_mock_logging_level(*level: bis_code_helpers.library_backend.MockLogger.LoggingLevels*)

Set the mock logger to a different logging level.

> **Supported levels are:**
>
> BISCodeHelpers.LoggingLevels.DEBUG
>
> BISCodeHelpers.LoggingLevels.INFO
>
> BISCodeHelpers.LoggingLevels.WARNING
>
> BISCodeHelpers.LoggingLevels.ERROR
>
> BISCodeHelpers.LoggingLevels.CRITICAL

> **Parameters**
>
> **level** – (BISCodeHelpers.LoggingLevels): Enumeration value for level.

> **Returns**
>
> None

`bis_code_helpers` supports logging in most if not all of it's functions, however there are two modes of logging available:

1. true logging, whereby you pass a `logging.Logger` instance to a function, and
2. mock logging, whereby you leave the *logger* argument blank in a function and messages that would usually be logged will be printed.

The mock logger defaults, like a true logger, to INFO level.

True Logging

bis_code_helpers.setup_logging(*log_folder: str, unique_log_name: str, logging_level: str*) → logging.Logger

Set up a logger with a standardised logging format.

> **Parameters**
>
> - **log_folder** – (str): Folder to hold log files.
> - **unique_log_name** – (str): Log file name.
> - **logging_level** – (str): Level for logger.
>
> **Returns**
>
> (logging.Logger): A fully set up logger.

Connection Management

bis_code_helpers.create_engine(*username: str, password: str, database: str, logger: logging.Logger = None*)

Sets up a database connection engine used to execute queries.

> **Parameters**
>
> - **username** – (str): Username for DB.
> - **password** – (str): Password for DB.
> - **database** – (str): DB address.
> - **logger** – (logging.Logger): Logger for logging debug and error messages.
>
> **Returns**
>
> (sqlalchemy.engine): DB connection engine.

The `create_engine` function is a simple wrapper on top of sqlalchemy's create engine function that handles the Oracle connection string formatting.

class **bis_code_helpers.ConnectionManager**(*engine*)

Context manager class to open and close connections as required.

> **Parameters**
>
> **engine** – (sqlalchemy.engine): Engine for connection.

The `ConnectionManager` class is used as such:

```
with ConnectionManager(engine) as conn:
    ...
```

Database Interaction

bis_code_helpers.check_existence_of_table(*table_name: str, engine, logger: logging.Logger = None*) → bool

Check existence of table on database.

> **Parameters**
>
> - **table_name** – (str): Name of table to perform operation on.
> - **engine** – (sqlalchemy.engine): DB engine used for DB connection.
> - **logger** – (logging.Logger): Logger to use for logging.
>
> **Returns**
>
> (bool): Existence of table.

bis_code_helpers.get_db_table_column_names(*table_name: str, engine, logger: logging.Logger = None*) → Optional[list]

Get column names of table on database. Checks for existence of table first.

> **Parameters**
>
> - **table_name** – (str): Name of table to perform operation on.
> - **engine** – (sqlalchemy.engine): DB engine used for DB connection.
> - **logger** – (logging.Logger): Logger to use for logging.
>
> **Returns**
>
> (Optional[list]): List of column names, None if table does not exist.

bis_code_helpers.get_db_table_row_count(*table_name: str, engine, logger: logging.Logger = None*) → Optional[int]

Get row count of table on database. Checks for existence of table first.

> **Parameters**
>
> - **table_name** – (str): Name of table to perform operation on.
> - **engine** – (sqlalchemy.engine): DB engine used for DB connection.
> - **logger** – (logging.Logger): Logger to use for logging.
>
> **Returns**
>
> (int): Number of rows, None if table does not exist.

bis_code_helpers.truncate_table(*table_name: str, engine, logger: logging.Logger = None*) → None

Truncate staging or prod table. Checks for existence of table first.

Parameters

- **table_name** – (str): Name of table to perform operation on.
- **engine** – (sqlalchemy.engine): DB engine used for DB connection.
- **logger** – (logging.logger): Logger to use for logging

Returns

None

---

bis_code_helpers.create_table(*data_results: pandas.core.frame.DataFrame, table_name: str, engine, allow_nulls: bool = True, logger: logging.Logger = None*) → None

Create a table based on the given DataFrame, automatically choosing data types.

Parameters

- **data_results** – (pd.DataFrame): Data to use for generating column names and data types.
- **table_name** – (str): Name of table to perform operation on.
- **engine** – (sqlalchemy.engine): DB engine used for DB connection.
- **allow_nulls** – (bool): Allow nulls in table.
- **logger** – (logging.logger): Logger to use for logging.

Returns

None

---

bis_code_helpers.drop_table(*table_name: str, engine, logger: logging.Logger = None*) → None

Drop table. Checks for existence of table first.

Parameters

- **table_name** – (str): Name of table to perform operation on.
- **engine** – (sqlalchemy.engine): DB engine used for DB connection.
- **logger** – (logging.logger): Logger to use for logging.

Returns

None

---

bis_code_helpers.upload_data_to_table(*table_data: pandas.core.frame.DataFrame, upload_partition_size: int, table_name: str, engine, logger: logging.Logger = None*) → None

Upload data in table_data DataFrame to table.

Parameters

- **table_data** – (pandas.DataFrame): data to be uploaded.
- **upload_partition_size** – (int): Number of rows to upload at a time.
- **table_name** – (str): Name of table to perform operation on.
- **engine** – (sqlalchemy.engine) DB engine used for DB connection.
- **logger** – (logging.Logger): Logger to use for logging.

Returns

None

---

bis_code_helpers.update_column_by_value(*old_value: int, new_value: int, table_name: str, column_name: str, engine, logger: logging.Logger = None*) → None

Update all rows in the production DB that have the old value in latest_prediction to have the new value.

Parameters

- **old_value** – (int): Value to select rows by.
- **new_value** – (int): Value to replace old value.
- **table_name** – (str): Name of table to perform operation on.
- **column_name** – (str): Name of column to update.
- **engine** – (sqlalchemy.engine): DB engine used for DB connection.
- **logger** – (logging.logger): Logger to use for logging.

Returns

None

---

bis_code_helpers.execute_select_query_on_db(*query: str, success_msg: str, error_msg: str, engine, logger: logging.Logger = None*) → pandas.core.frame.DataFrame

Execute a returning select query.

Parameters

- **query** – (str): Query to be executed.
- **success_msg** – (str): Debug message for successful execution.
- **error_msg** – (str): Error message for failed execution.
- **engine** – (sqlalchemy.engine): DB engine used for DB connection.
- **logger** – (logging.Logger): Logger to use for logging.

Returns

(pandas.Dataframe): Date returned from DB.

bis_code_helpers.execute_action_query_on_db(*query: str, success_msg: str, error_msg: str, engine, logger: logging.Logger = None*) → None

Execute a non-returning, commit required query.

> **Parameters**
>
> - **query** – (str): Query to be executed.
> - **success_msg** – (str): Debug message for successful execution.
> - **error_msg** – (str): Error message for failed execution.
> - **engine** – (sqlalchemy.engine): DB engine used for DB connection.
> - **logger** – (logging.Logger): Logger to use for logging.
>
> **Returns**
>
> None

Logged Exceptions

These are merely exceptions that log their messages.

All of the functions in this package use these over their non-logged counterparts.

exception **bis_code_helpers.LoggedValueError**(*logger: logging.Logger, message: str*)

ValueError, but with builtin logging.

> **Parameters**
>
> - **logger** – (logging.Logger): Logger for logging.
> - **message** – (str): Exception message to display.

exception **bis_code_helpers.LoggedDataError**(*logger: logging.Logger, message: str*)

DataError, but with builtin logging.

> **Parameters**
>
> - **logger** – (logging.Logger): Logger for logging.
> - **message** – (str): Exception message to display.

exception **bis_code_helpers.LoggedDatabaseError**(*logger: logging.Logger, message: str*)

DatabaseError, but with builtin logging.

> **Parameters**
>
> - **logger** – (logging.Logger): Logger for logging.

- **message** – (str): Exception message to display.

---

exception **bis_code_helpers.LoggedSubprocessError**(*logger: logging.Logger*, *message: str*)

SubprocessError, but with builtin logging.

> **Parameters**
>
> - **logger** – (logging.Logger): Logger for logging.
> - **message** – (str): Exception message to display.