Cosmic Frog Python Library

Description

This library provides helper functions for working with data in Cosmic Frog.

The following documentation explains how to connect to Cosmic Frog models and use this to create your own applications.

Use cases

The main purpose of the library is to facilitate reading, writing and modifying Cosmic Frog model tables, while minimizing the amount of code that is required.

As an example, to connect to an existing Cosmic Frog model you can do:

```
model = FrogModel("my model name")
```

And then to read a table of data from the model you can do this:

```
data = model.read_table("Customers")
```

Once the data has been read it can be manipulated with the full power of the python Pandas module, for efficient data querying and updates, before being written back with a similarly brief call:

```
model.write_table("Customers", data)
```

Full Pandas documentation can be found via the web: Pandas documentation

Installation

The cosmicfrog library is installed using pip:

```
pip install cosmicfrog
```

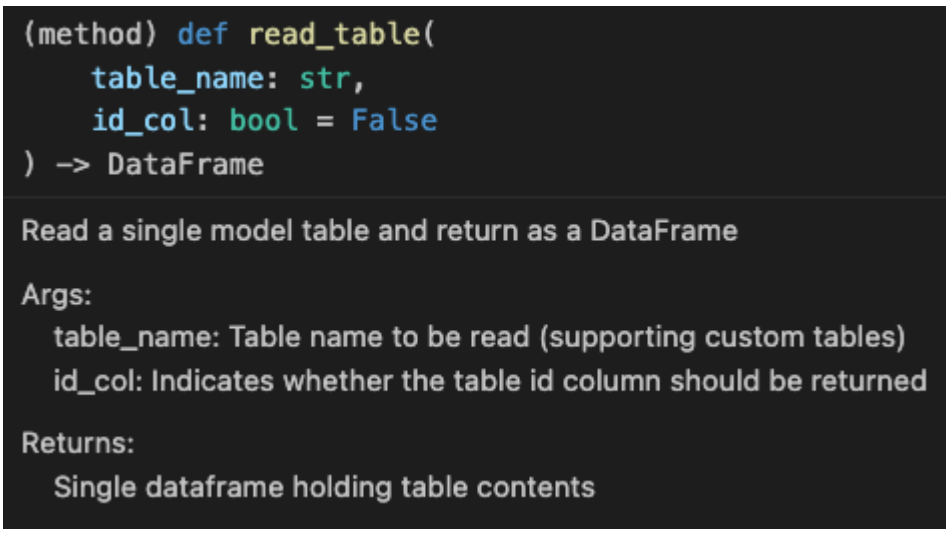
It can then be referenced by including the FrogModel helper class in your code:

```
from cosmicfrog import FrogModel
```

Working with the library

The library is designed to be easy to use, and to handle most boilerplate code for you - leaving you to concentrate on modelling and data manipulation.

For many python editors, you can get help on each function via tooltips:



Creating a FrogModel with python

A class called FrogModel is the main helper class in the cosmicfrog library. It will allow you to connect to a Cosmic Frog model and interact with it directly.

In order to connect to a model you will need to authenticate yourself. This is done when the FrogModel is created, and can happen a number of ways:

Running python on the Optilogic platform

When you are running python code on the Optilogic platform, the cosmicfrog library will auto-detect your login credentials, and models can be opened simply by using the model name:

```
model = FrogModel("MyExample")
```

The model is now ready to access data within the "MyExample" model. The name here is the same name you will see when opening the model in the Cosmic Frog UI.

Working on the desktop

When you are running python code locally, the cosmicfrog library will not be able to auto-detect your login credentials, and an App key should be supplied.

An App key is a key that encapsulates your login credentials.

> ⚠️ **Warning:** It is important to keep App keys secure and to not distribute them to anyone that you do not wish to access your CosmicFrog data.

If you want to recall an App key this can be done by deleting the App key in the UI, which will mean that the key will become invalid and no longer authenticate.

The UI for creating App keys can be found on the User Account Page.

Once you have created an App key, copy it and paste it into a file called "app.key" that is placed alongside the python script you are running. When your code is run, the cosmicfrog library will read this file and use the App key in the file for authentication.

Working on the desktop (alternative)

In some scenarios it may not be possible to create an app.key file or to place it in the correct place. It is also possible to pass the app key directly when creating a model like this:

```
model = FrogModel("MyExample", app_key="my_app_key_xyz123")
```

> ⚠️ **Warning:** Please be aware, when hardcoding an App key this way, if you give others access to this script they may use it to access your Cosmic Frog data. We recommend that when sharing a script with others, app keys are *not* included.

Firewalls for desktop access

> 💡 **Tip:** All Cosmic Frog models are protected by a network firewall. If you are having difficulty accessing your models, check that you have opened access via the firewall. This can be done via the UI here:

Optilogic Storage Dashboard

Using the firewall tab, ensure that your desktop IP is added to the exclusion list in order to allow your local python script access to your models.

Reading data from a single table

Data in a Cosmic Frog model is held in tables which make up the Anura supply chain modelling schema.

The easy way to get the data from a table is via the `read_table` function. Once you have connected the FrogModel (see above) call the function with the required table name as follows:

```
data = model.read_table("Customers")
```

The data is returned as a Pandas DataFrame which can now be seen directly:

```
print(data)
```

It may also be manipulated using the python Pandas module (see above for documentation link).

By default the data is returned without the ID column. This is the recommended way to manipulate the data, leaving ID handling to the library.

If you have a special case where the ID column is required, you can have it included like this:

```
data = model.read_table("Customers", id_col = True)
```

Reading data from multiple tables

In addition to reading single tables, you may also want to download the data for multiple tables in a single call.

To do this, first create a list of the tables you need, and then call the `read_tables` function:

```
table_list = ["Suppliers", "Facilities", "Customers"]
tables = model.read_tables(table_list)
print(tables["Customers"])
```

The return from `read_tables` is a python dictionary, with the table name as key, and the table content in a Pandas DataFrame as the value.

This function also supports the `id_col` parameter, and the parameter now affects all the tables being fetched:

```
tables = model.read_tables(table_list, id_col = True)
```

Writing data

Writing data back to a table is similarly easy to do, via the `write_table` function. Here you specify first the destination table, followed by a dataframe containing the data to be written:

```
model.write_table("Customers", my_data)
```

The rows in the DataFrame will be appended directly to the table.

If you would prefer to first clear the table and replace the data, then this can be done using the `overwrite` parameter:

```
model.write_table("Customers", my_data, overwrite = True)
```

> 💡 **Tip:** To update only some rows in a table, which can be useful when a large amount of data exists, the Upsert function may be useful.

Writing data to multiple tables

Data can be written to multiple tables in a single operation using the `write_tables` function. Here the table data is held in a dictionary, indexed by table name.

This is the format that is returned by the `read_tables` function, making it easy to read a set of tables, modify these, and then write back, or write to another model.

```
model.write_tables(my_tables_data, overwrite = True)
```

As with `write_table`, `write_tables` accepts an `overwrite` parameter. This parameter setting is applied to all tables being written.

Updating data (Upsert)

When updating large tables, or making incremental changes, it is sometimes useful to update existing rows, as well as inserting new rows.

This is accomplished in Cosmic Frog via the upsert functions, which use the keys defined in the table to determine matches, and then act accordingly.

The Upsert operation proceeds as follows:

1. For each row in the input data, the target table is scanned for matches where data in the table 'key columns' is matching.
2. For the matches, the data in the table is updated (non key columns are changed to match the input data).
3. For rows in the input data that do not match any existing row, a new row is added.

This combines update and insert, and is known as an *upsert* operation.

The key columns for each table are defined in the Anura Schema.

> **Tip:** You can add your own table keys in addition to the existing key columns by adding custom columns and setting them to be keys in the Cosmic Frog UI. This allows you to further control the behaviour of upsert.

Upsert from csv file

To upsert a csv file into a table, call `upsert_csv`, passing the target table and the filename of the .csv file to be used:

```
model.upsert_csv("Customers", "customers.csv")
```

Upsert from Excel file

To upsert an .xlsx file into a table, call `upsert_excel`, passing the filename of the .xlsx file only:

```
model.upsert_excel("tables_to_upsert.xlsx")
```

> **Note:** When upserting Excel files the upsert function will use the names of worksheets to determine which table the data is intended for (e.g. the *Customers* worksheet will be upserted into the *Customers* Cosmic Frog table).

Clearing a table

To clear a table (removing all rows of data) use the `clear_table` function:

```
model.clear_table("Customers")
```

> ⚠️ **Warning:** Once a table is cleared it cannot be undone.

Executing SQL

As well as offering pre-defined operations, like fetching and writing datatables, the library also makes it easy to run arbitrary sql against a model to perform any custom actions or analysis required.

> 💡 **Tip:** Cosmic Frog models can be accessed using the Postgres dialect of SQL

Fetching data using an arbitrary sql statement

To run a SELECT SQL statement that returns data:

```
data = model.read_sql("SELECT COUNT(1) FROM CUSTOMERS")
```

The sql will be executed within the model, and the result is returned as a Pandas DataFrame.

Modifying data using an arbitrary sql statement

To run a command that alters data, but does not need to return a result (e.g. UPDATE, DELETE):

```
model.exec_sql("DELETE FROM CUSTOMERS")
```

Geocoding table data

For specific tables that have a geolocation (e.g. Suppliers, Facilities, Customers) the library can be used to trigger geocoding for the table:

```
model.geocode_table("Customers")
```

Default geocoding provider will be *MapBox* and it will happen automatically. For other providers a `geoapikey` should be provided:

```
model.geocode_table("Customers", geoprovider="Google",
geoapikey="123_my_google_key")
```

When geocoding with *MapBox* you can opt to ignore low confidence results (default), or to have them included. This is achieved using the following parameter:

```
model.geocode_table("Customers", "MapBox", ignore_low_confidence = False)
```

The behaviour of this parameter is specific to each API, please contact support for more detail.

When calling `geocode_table` with `fire_and_forget=False`, the function will wait until geocoding completes before returning:

```
geocode_status = model.geocode_table('facilities', fire_and_forget=False)
```

Querying Anura schema tables

To get a list of tables in the model the `get_tablelist` function can be used:

```
tables = model.get_tablelist()
```

The list returned will contain all tables in the schema by default.

To get a specific list of tables, the following parameters can be specified:

1. `input_only` returns only Input tables
2. `output_only` returns only Output tables
3. `technology_filter` specifies an engine technology

When specifying a technology filter the options are based on the Cosmic Frog technology names for each engine:

- "NEO" - Optimization engine
- "THROG" - Simulation engine
- "TRIAD" - Risk engine
- "DART" - Greenfield engine
- "HOPPER" - Transportation engine

Example:

```
tables = model.get_tablelist(input_only = True, technology_filter =
"DART")
```

This will return only input tables for the Greenfield engine.

By default, table names returned will be lower cased, for ease of use with other library functions.

> 📝 **Note:** To see the original table name as shown in Cosmic Frog, use the parameter `original_names = True`.

Querying Anura schema columns

The library can be used to fetch details about the Anura supply chain modelling schema.

For instance, to get the list of columns for an Anura table:

```
column_list = model.get_columns("Customers")
```

This will return the columns specified in the Anura schema.

> **Note:** The list returned is taken from the schema, rather than the model - so if custom columns have been added to this particular model they will not be returned. To retrieve the actual columns for a specific model table, use `get_table_columns_from_model()`.

Querying actual table columns

To query a specific model for all columns in a table, including custom columns that have been added, the code is:

```
actual_cols = model.get_table_columns_from_model("Customers")
```

By default this will exclude the `id` column, but to include this also add the `id_col` parameter:

```
actual_cols = model.get_table_columns_from_model("Customers", id_col =
True)
```

Additional functionalities

In addition to the above operations for reading/writing data, the `FrogModel` class provides several advanced features for model management, scenario runs, run parameters, and custom schema definitions (tables and columns). Below is an overview of these extended functionalities, along with code examples.

Model Management

You can manage models directly using static methods on `FrogModel` (which do not require an instantiated `FrogModel` object) as well as instance methods once you have created a `FrogModel` object.

1. **Get All Available Model Templates for new model creation**

```
templates = FrogModel.all_available_model_templates()
```

2. **Get All Models**

```
all_models = FrogModel.all_models()
```

3. **Create a New Model**

   On create method will return model instance

```
model = FrogModel.create_model("NewModelName")
```

4. **Edit Model**

```
renamed_model = model.edit_model("NewModelName")
```

5. **Delete Model**

   Option without model initalization:

```
FrogModel.delete_model("Model-name-for-deletion")
```

   Initialized model:

```
delete_result = model.delete()
```

6. **Share Model**

```
share_result = model.share("other-user-username-or-email-address")
```

7. **Remove Share Access**

```
remove_result = model.remove_share_access("other-user-username-or-
email-address")
```

8. **Clone Model**

```
clone_result = model.clone("MyModelClone")
cloned_model = FrogModel("MyModelClone")
```

9. **Archive a Model**

```
archive_result = model.archive()
```

10. **Restore Archived Model**

```
restore_result = FrogModel.archive_restore("MyArchivedModel")
```

11. **Get All Archived Models**

```
archived_models = FrogModel.archived_models()
```

Scenario Management

The library supports running and managing scenarios within your Cosmic Frog models.

1. **Preview Scenarios**

   You can see a list of all scenarios inside of a model, their technologies(engines), items with their conditions actions and table names.

   ```
   scenarios = model.all_scenarios_preview()
   ```

2. **Run a Scenario**

   Run scenarios within a model. Can run all scenarios in a model by passing ["All"]. If scenario names are not passed, will run the Baseline scenario (default). Scenario names are case sensitive.

   If engine is not passed, will run on the default engine from the scenario table (technology column). If technology is not set in the scenario table, will default to "neo". If technology is set to unknown value, will return an error.

   By default function will run the scenario on resource size S, which can be changed by passing resource_size parameter.

   By default function will await all of the scenarios to complete. Pass `fire_and_forget=True` to run in the background.

   If not sure about configuration before running, pass `check_configuration_before_run=True` to return the final config without actually running anything.

   If running with infeasibility check, set run_neo_with_infeasibility=True; engine parameter will be ignored and set to "neo".

```
# Run the default "Baseline" scenario with NEO engine on s resource
and await for it to finish
run_info = model.run_scenario()

# Run a specific scenarios with some extra parameters and do not await
for it to finish
run_info = model.run_scenario(
   scenarios=["No Detroit DC", "No Flow Constraint"],
   engine="neo",
   resource_size="m",
   fire_and_forget=True
)

# Run all scenarios
run_info = model.run_scenario(
   scenarios=['All']
)
```

3. **Run multiple scenarios with different configuration**

   Run multiple scenarios with different engine and resource size

```
scenarios_with_custom_configuration = [
   {
      "scenario_name": "Baseline",
      "engine": "neo",
      "resource_size": "s",
   },
   {
      "scenario_name": "No Detroit DC",
      "engine": "neo",
      "resource_size": "s",
   },
   {
      "scenario_name": "GF 9 Facilities",
      "engine": "neo",
      "resource_size": "m",
   },
   {
      "scenario_name": "Throg Example",
      "engine": "throg",
      "resource_size": "m",
   },
]

response =
frog_model.run_multiple_scenarios_with_custom_configuration(

scenarios_with_custom_configuration=scenarios_with_custom_configuratio
```

```
n
)
```

4. **Stop a Running Scenario**

   You can stop a scenario either by its name or by job key:

```
# By scenario name
stop_result = model.stop_scenario(scenario_name="No Detroit DC")

# By job key
stop_result = model.stop_scenario(job_key="xxxxx-xxx-xxxx-xxxx-
xxxxxxxx")
```

5. **All Currently Running Scenarios**

```
jobs = model.all_running_scenarios()
```

6. **Check Scenario Status**

   Similar to stopping a scenario, you can check status by name or by job key:

```
# By scenario name
status_info = model.check_scenario_status(scenario_name="No Detroit
DC")

# By job key
status_info = model.check_scenario_status(job_key="xxxxx-xxx-xxxx-
xxxx-xxxxxxxx")
```

7. **Check Scenario Run Logs**

   Get the logs of a certain job:

```
job_logs = model.get_job_logs(job_key='xxxxx-xxx-xxxx-xxxx-xxxxxxxx')
```

8. **Check Scenario Run Error Logs**

```
job_error = model.get_scenario_run_error_log(job_key='xxxxx-xxx-xxxx-
xxxx-xxxxxxxx')
```

9. **Get All Jobs From Solver Job**

```
jobs_from_solver = model.get_all_jobs_for_solver_job(job_key='xxxxx-
xxx-xxxx-xxxx-xxxxxxxx')
```

10. **Tail Records For Running Scenario**

```
live_records = frog_model.tail_job_records(job_key="xxxxx-xxx-xxxx-xxxx-
xxxxxxxx")
```

11. **Get Job Records**

```
job_records = model.job_records(job_key="xxxxx-xxx-xxxx-xxxx-xxxxxxxx")
```

Scenario Run Parameters Management (ModelRunOptions table)

Cosmic Frog scenarios can have run parameters (model run options). The library supports viewing, creating, editing, and deleting these parameters.

1. **Get All Run Parameters**

```
# Get all run parameters relevant to NEO engine, engine is optional
run_params = model.get_all_run_parameters(engine="neo")
```

2. **Update a Run Parameter Value**

```
update_result =
model.update_run_parameter_value("NumberOfReplications", "1")
```

3. **Add a New Run Parameter**

```
# The parameter dictionary typically has an 'option' and 'value'
new_param = {
   "option": "MyNewParam",
   "value": "SomeValue"
}
add_result = model.add_run_parameter(new_param)
```

4. **Delete a Run Parameter**

```
delete_result = model.delete_run_parameter("MyNewParam")
```

Custom Tables and Columns Management

In addition to the built-in Anura schema tables, you can create your own custom tables and columns within a model.

1. **List All Custom Tables**

```
tables = model.get_all_custom_tables()
```

2. **Create a New Custom Table**

```
create_table_result = model.create_table("MyCustomTable")
```

3. **Delete a Custom Table**

```
delete_table_result = model.delete_table("MyCustomTable")
```

4. **Rename a Custom Table**

```
rename_table_result = model.rename_table("MyCustomTable",
"NewTableName")
```

5. **List All Custom Columns**

```
columns = model.get_all_custom_columns("MyCustomTable")
```

6. **Create a New Custom Column**

```
create_column_result = model.create_custom_column(
    'MyCustomTable',
    'new_column',
    data_type='text',
    key_column=False
)
```

7. **Delete a Custom Column**

```
delete_column_result = model.delete_custom_column("MyCustomTable",
"new_column")
```

8. **Edit an Existing Custom Column**

```
edit_column_result = model.edit_custom_column(
    "MyCustomTable",
    "new_column",
    "new_column_name"
)
```

9. **Bulk Create Custom Columns**

   Bulk create custom columns.

   Each column should be a dictionary with the following keys:

   - table_name: str
   - column_name: str
   - data_type: str (Optional) - Default is 'text'
   - key_column: bool (Optional) - Default is False
   - pseudo: bool (Optional) - Default is True

```
bulk_custom_columns_result = model.bulk_create_custom_columns(
   [
      {
         "table_name": "Facilities",
         "column_name": "Custom1",
         "data_type": "integer",
      },
      {
         "table_name": "Facilities",
         "column_name": "Custom2"
      },
      {
         "table_name": "Customers",
         "column_name": "Custom3",
      }
   ]
)
```

Happy modelling! If you have any questions or need help with any of these features, feel free to reach out to your support team or consult the Optilogic documentation.