grist_api

Release 0.1

Grist Labs

Aug 08, 2021

MODULE DOCUMENTATION:

The `grist_api` module is a Python client library for interacting with Grist.

CHAPTER

ONE

INSTALLATION

grist_api is available on PyPI: https://pypi.python.org/pypi/grist_api/:

```
pip install grist_api
```

The code is on GitHub: https://github.com/gristlabs/py_grist_api.

The API Reference is here: https://py-grist-api.readthedocs.io/en/latest/grist_api.html.

CHAPTER

TWO

USAGE

See `tests/test_grist_api.py` for usage examples. A simple script to add some rows to a table and then fetch all cells in the table could look like:

```
from grist_api import GristDocAPI
import os

SERVER = "https://subdomain.getgrist.com"          # your org goes here
DOC_ID = "9dc7e414-2761-4ef2-bc28-310e634754fb"    #  document id goes here

# Get api key from your Profile Settings, and run with GRIST_API_KEY=<key>
api = GristDocAPI(DOC_ID, server=SERVER)

# add some rows to a table
rows = api.add_records('Table1', [
    {'food': 'eggs'},
    {'food': 'beets'}
])

# fetch all the rows
data = api.fetch_table('Table1')
print(data)
```

CHAPTER

THREE

TESTS

Tests are in the `tests/` subdirectory. To run all tests, run:

```
nosetests
```

3.1 grist_api module

Client-side library to interact with Grist.

Handling data types. Currently, datetime.date and datetime.datetime objects sent to Grist (with add_records() or update_records()) get converted to numerical timestamps as expected by Grist.

Dates received from Grist remain as numerical timestamps, and may be converted using ts_to_date() function exported by this module.

class grist_api.**ColSpec**(*gcol*, *ncol*, *gtype*)
Bases: *grist_api.ColSpec*

Column specifier for syncing data. Each column is represented by the tuple *(grist_col_id, new_data_col_id[, grist_type])*.

class grist_api.**GristDocAPI**(*doc_id*, *api_key=None*, *server='https://api.getgrist.com'*, *dryrun=False*)
Bases: `object`

Class for interacting with a Grist document.

add_records(*table_name*, *record_dicts*, *chunk_size=None*)
Adds new records to the given table. The data is a list of dictionaries, with keys corresponding to the columns in the table. Returns a list of added rowIds.

If chunk_size is given, we'll make multiple requests, each limited to chunk_size rows.

call(*url*, *json_data=None*, *method=None*, *prefix=None*)
Low-level interface to make a REST call.

delete_records(*table_name*, *record_ids*, *chunk_size=None*)
Deletes records from the given table. The data is a list of record IDs.

fetch_table(*table_name*, *filters=None*)
Fetch all data in the table by the given name, returning a list of namedtuples with field names corresponding to the columns in that table.

If filters is given, it should be a dictionary mapping column names to values, to fetch only records that match.

sync_table(*table_id*, *new_data*, *key_cols*, *other_cols*, *grist_fetch=None*, *chunk_size=None*, *filters=None*)
Updates Grist table with new data, updating existing rows or adding new ones, matching rows on the given key columns. (This method does not remove rows from Grist.)

New data is a list of objects with column IDs as attributes (e.g. namedtuple or sqlalchemy result rows).

Parameters key_cols and other_cols list primary-key columns, and other columns to be synced. Each column in these lists must have the form (grist_col_id, new_data_col_id[, opt_type]). See make_type() for available types. In place of grist_col_id or new_data_col_id, you may use a function that takes a record and returns a value.

Initial Grist data is fetched using fetch_table(table_id), unless grist_fetch is given, in which case it should contain the result of such a call.

If chunk_size is given, individual requests will be limited to chunk_size rows each.

If filters is given, it should be a dictionary mapping grist_col_ids from key columns to values. Only records matching these filters will be synced.

update_records(*table_name*, *record_dicts*, *group_if_needed=False*, *chunk_size=None*)
Update existing records in the given table. The data is a list of dictionaries, with keys corresponding to the columns in the table. Each record must contain the key "id" with the rowId of the row to update.

If records aren't all for the same set of columns, then a single-call update is impossible. With group_if_needed is set, we'll make multiple calls. Otherwise, will raise an exception.

If chunk_size is given, we'll make multiple requests, each limited to chunk_size rows.

grist_api.**chunks**(*items*, *max_size=None*)
Generator to return subsets of items as chunks of size at most max_size.

grist_api.**date_to_ts**(*date*)
Converts a datetime.date object to a numerical timestamp of the UTC midnight in seconds.

grist_api.**desc_col_values**(*data*)
Returns a human-readable summary of the given TableData object (dict mapping column name to list of values).

grist_api.**dt_to_ts**(*dtime*)
Converts a datetime.datetime object to a numerical timestamp in seconds. Defaults to UTC if dtime is unaware (has no associated timezone).

grist_api.**make_type**(*value*, *grist_type*)
Convert a value, whether from Grist or external, to a sensible type, determined by grist_type, which should correspond to the type of the column in Grist. Currently supported types are:

- Numeric: empty values default to 0.0
- Text: empty values default to ""
- Date: in Grist values are numerical timestamps; in Python, datetime.date.
- DateTime: in Grist values are numerical timestamps; in Python, datetime.datetime.

grist_api.**ts_to_date**(*timestamp*)
Converts a numerical timestamp in seconds to a datetime.date object.

grist_api.**ts_to_dt**(*timestamp*)
Converts a numerical timestamp in seconds to a naive datetime.datetime object representing UTC.

PYTHON MODULE INDEX

INDEX