CollectionBatchTool Documentation

Release 0.1.6

Markus Englund

May 24, 2016

Contents

1 Project background **3**
- 1.1 Installing CollectionBatchTool . . . 3
- 1.2 Quickstart . . . 3
- 1.3 Advanced Operations . . . 7
- 1.4 Preparing CSV files . . . 11
- 1.5 Supported Specify Tables . . . 12
- 1.6 API Reference . . . 13

2 Indices and tables **21**

Python Module Index **23**

CollectionBatchTool is a Python library for importing, exporting, and updating batches of collection data in Specify. The intended audience is advanced users such as data managers, migration specialists, and system administrators.

- Built on top of the packages peewee and pandas
- Fast uploading of large datasets
- Requires no prior knowledge in SQL and little knowledge of Python

Source repository: https://github.com/jmenglund/CollectionBatchTool

New to CollectionBatchTool? Here are a few documents to help you get started:

- *Quickstart guide* – covers the most basic stuff and will take you about 10 minutes to go through.
- *Preparing CSV files* – explains how to prepare files for data import.
- *Supported Specify Tables* – lists the database tables currently supported.

Important: The current version of CollectionBatchTool (0.1.6) only supports Python 3.

CHAPTER 1

Project background

CollectionBatchTool has been developed within the DINA-project in order to support migration of large datasets (100,000+ records) to Specify. The tool was first released in September 2015.

1.1 Installing CollectionBatchTool

For most users, the easiest way is probably to install the latest version hosted on PyPI:

```
$ pip install collectionbatchtool
```

The project is hosted at https://github.com/jmenglund/CollectionBatchTool and can also be installed using git:

```
$ git clone https://github.com/jmenglund/CollectionBatchTool.git
$ cd CollectionBatchTool
$ python setup.py install
```

Tip: You may consider installing CollectionBatchTool (and its required packages) within a virtual environment in order to avoid cluttering your system's Python path. See for example the package virtualenv.

1.2 Quickstart

This guide gives you a brief introduction to CollectionBatchTool's most basic features. The following will be covered:

- *Before you begin*
- *Setting up the database connection*
- *Exporting data from a single table*
- *Importing data to a single table*

- *Updating existing database records*

1.2.1 Before you begin

A few things need to be in place in order to be able to use CollectionBatchTool:

- You need root access to the Specify (MySQL) database you want to work with.
- Your collection must be present in the database. If the collection is missing, you have to create it by using *Specify Setup Wizard*.
- Your Specify user must be available to the collection in question.

Warning: Using CollectionBatchTool is equivalent to writing SQL-statement against the database as a root user. You are responsible for any changes made to the data. In order to avoid concurrency problems, make sure that no one else is accessing the database while you are using the tool.

1.2.2 Setting up the database connection

CollectionBatchTool lets you work with one Specify collection at a time. Enter your personal settings in a configuration file similar to what is shown below. Then, save the file with a suitable name, for example `settings.cfg`.

```
[MySQL]
Database = my_database
Host = localhost
User = root
Password = password

[Specify]
CollectionName = My collection
User = user
```

To apply your settings, use the function *apply_user_settings()* provided with the path to your configuration file:

```
>>> from collectionbatchtool import *
>>> apply_user_settings('settings.cfg')
```

Note: It is good practice to always backup your data before you start working with a database.

1.2.3 Exporting data from a single table

In this exercise we will export data from the agent-table in Specify to a comma-separated values (CSV) file. We assume that the database connection has been set up as described in the previous section. Now, let's begin by first creating an *AgentDataset* object:

```
>>> agt = AgentDataset()
```

This newly created object will eventually hold all the data we want to export. For the moment, it basically contains an empty dataframe stored to the *frame* attribute. You can verify that the dataframe is empty by examining the *AgentDataset* object (see the last line of the output):

```
>>> agt
<class 'collectionbatchtool.AgentDataset'>
model: <class 'specifymodels.Agent'>
key_columns: {
    'agentid': 'agent_sourceid'
    'createdbyagentid': 'createdbyagent_sourceid'
    'modifiedbyagentid': 'modifiedbyagent_sourceid'
    'parentorganizationid': 'parentorganization_sourceid'}
static_content: {
    'divisionid': 2
    'specifyuserid': None}
where_clause: <class 'peewee.Expression'>
frame: <class 'pandas.core.frame.DataFrame'> [0 rows x 30 columns]
```

You can also take a look at the *frame* attribute directly:

```
>>> agt.frame
Empty DataFrame
Columns: [
    agent_sourceid, createdbyagent_sourceid, modifiedbyagent_sourceid,
    parentorganization_sourceid, abbreviation, agenttype, dateofbirth,
    dateofbirthprecision, dateofdeath, dateofdeathprecision, datetype, email,
    firstname, guid, initials, interests, jobtitle, lastname, middleinitial,
    remarks, suffix, timestampcreated, timestampmodified, title, url, version,
    agentid, createdbyagentid, modifiedbyagentid, parentorganizationid]
Index: []

[0 rows x 30 columns]
```

Next, we would like to load data from the agent table into our object's *frame* attribute. We do so by using the *from_database()* method. By setting `quiet=False` we will get some information on what's going on:

```
>>> agt.from_database(quiet=False)
[AgentDataset] reading database records: 1/1
```

If there are any agent-records associated with your collection, these should now be stored to your *AgentDataset* object. A new collection within its own division in Specify contains just a single agent-record (as in our example).

Writing the data to a CSV file is just as easy as retrieving the data from the database. The method *to_csv()* and a file path is all that is needed:

```
>>> agt.to_csv('agent.csv', update_sourceid=True, quiet=False)
[AgentDataset] updating SourceID-columns...
    copying 'agentid' to 'agent_sourceid' [1 value]
    copying 'createdbyagentid' to 'createdbyagent_sourceid' [0 values]
    copying 'modifiedbyagentid' to 'modifiedbyagent_sourceid' [0 values]
    copying 'parentorganizationid' to 'parentorganization_sourceid' [0 values]
[AgentDataset] writing to CSV file...
    1 rows x 26 columns; agent.csv
```

In the example above, we use the `update_sourceid` parameter to ensure that every ID-column is copied to its corresponding SourceID-column before the data is written to the file.

Note: Want to export data from some other table? Take a look at *supported Specify tables* to see which tables are currently available.

1.2.4 Importing data to a single table

Data import is not so different from data export. One important difference, though, is that you first need to prepare CSV files according to some specific rules. We don't go into the details here, but if you wish you can read about the format specifications in the document on *how to prepare CSV files*.

In this excercise we will import data to the agent-table. Like in the export example above, we assume that the database connection has been set up properly. We will try to import a small sample dataset (three columns and ten rows) listing some of the Apostles of Linnaeus. The first column, *agent_sourceid*, is somewhat special and may be used to connect records outside of the Specify database. This column can also be used to trace imported data, as we will see towards the end of this exercise. The sample file `apostles.csv` contains the following records:

agent_sourceid	firstname	lastname
1	Pehr	Osbeck
2	Peter	Forsskål
3	Pehr	Löfling
4	Pehr	Kalm
5	Daniel	Rolander
6	Johan Peter	Falck
7	Daniel	Solander
8	Carl Peter	Thunberg
9	Anders	Sparrman
10	Peter Jonas	Bergius

`apostles.csv`

Like with exports, we start out by creating an *`AgentDataset`* object:

```
>>> agt = AgentDataset()
```

To read the data from the CSV file, we use the *`from_csv()`* method:

```
>>> agt.from_csv('apostles.csv', quiet=False)
[AgentDataset] reading CSV file...
    10 rows x 3 columns; apostles.csv
```

The *`AgentDataset`* object should now hold the ten records. We continue with uploading the data to the agent-table. The method *`to_database()`* takes care of the upload for us. We use the method's `defaults` parameter to insert default values instead of `NULL`. This parameter accepts a python `dict` with column names and values to insert:

```
>>> agt_defaults = {
...     'agenttype': 1,
...     'dateofbirthprecision': 1,
...     'dateofdeathprecision': 1,
...     'middleinitial': ''
... }
>>> agt.to_database(defaults=agt_defaults, quiet=False)
[AgentDataset] loading records to database: 10/10
```

After the import is completed, your *`AgentDataset`* object will automatically get updated with the inserted records' primary key values. If you look at the *`frame`* attribute of your *`AgentDataset`* object, you should see a new value in the *agentid*-column for every record that was imported to the database.

Now, let's try to upload the dataset again. As you should notice, none of the records were inserted into the database this time. The reason is the new values in the *agentid*-column. Only records that lack a primary key value will get uploaded to the database.

Finally, we can use the *`write_mapping_to_csv()`* method to export the mapping between *agent_sourceid* and *agentid*. This mapping allow us to trace the source of every imported record.

```
>>> agt.write_mapping_to_csv('agent-mapping.csv')
```

Important: There is no data validation carried out by CollectionBatchTool prior to import. An incorrect datatype, or violation of a database contraint, will result in an error and an exception being raised.

1.2.5 Updating existing database records

In the last exercise of the quickstart guide, we will try to update some of the records that we imported *previously*. Suppose that we want to update the agent-table with new birthyears for three of the apostles that were imported.

The new sample file `apostles_birthyear.csv` contains the following information:

firstname	lastname	dateofbirth	dateofbirthprecision
Peter	Forsskål	1732-01-01	3
Daniel	Solander	1733-01-01	3
Carl Peter	Thunberg	1743-01-01	3

`apostles_birthyear.csv`

We begin by creating a new *`AgentDataset`* object and reading the new sample data into that object:

```
>>> agt = AgentDataset()
>>> agt.from_csv('apostles_birthyear.csv', quiet=False)
[AgentDataset] reading CSV file...
    3 rows x 4 columns; apostles_birthyear.csv
```

Next, we get primary key values from the agent-table based on content in the columns *firstname* and *lastname* (we assume that the combination of first and last names will uniquely identify individual agent-records). We use the *`match_database_records()`* method:

```
>>> agt.match_database_records(['firstname', 'lastname'], quiet=False)
[AgentDataset] updating primary key from database...
    target-column:   'agentid'
    match-column(s): ['firstname', 'lastname']
    matches:         3/3
```

You should now be able to see the updated values in the *agentid*-column if you check the *`frame`* attribute of your *`AgentDataset`* object. Once the primary key values are in place, it's easy to update the database with information from the two new columns by using the *`update_database_records()`* method:

```
>>> agt.update_database_records(['dateofbirth', 'dateofbirthprecision'], quiet=False)
[AgentDataset] updating database records: 3/3
```

You have now reached the end of the quickstart guide. If you want to learn more, you can continue to the guide on CollectionBatchTool's *advanced features* or read more about functions, classes and methods in the *API reference*.

1.3 Advanced Operations

This document demonstrates some of SpecifyBatchTool's more advanced features. The following sections are included:

- *Exporting data from multiple tables*
- *Importing data to multiple tables*

- *Counting database records*

It is assumed here that you are familiar with the basic operations described in the *quickstart guide*. Before you run any of the code in this document, you should initiate your database as described in the section *Setting up the database connection* in the quickstart guide.

1.3.1 Exporting data from multiple tables

Exporting data from multiple tables is easy as exporting data from a single table. Just use the *from_database()* and *to_csv()* methods. Remember to always set `update_sourceid=True` in the *to_csv()* method to ensure that individual records can be connected also after they have been exported. Here is an example of how you export data from the locality- and geography-tables.

```
>>> loc = LocalitytDataset()
>>> loc.from_database()
>>> loc.to_csv(update_sourceid=True)
>>> geo = GeographyDataset()
>>> geo.from_database()
>>> geo.to_csv(update_sourceid=True)
```

To export all the data available to CollectionBatchTool, we can write a simple command-line script:

```
#!/usr/bin/env python

"""
export_all_data.py - script for exporting available data.
"""

import os
import argparse

from collectionbatchtool import *


def export_all_data(
        config_file, output_dir=None,
        drop_empty_columns=False, quiet=True):
    """
    Export data from Specify to CSV files.

    Parameters
    ----------
    output_dir : str
        Path to the output directory.
    """
    apply_user_settings(config_file)
    output_dir = output_dir if output_dir else ''
    for tabledataset_subclass in TableDataset.__subclasses__():
        instance = tabledataset_subclass()
        if instance.database_query.count() > 0:  # no files without data
            instance.from_database(quiet=quiet)
            filename = instance.model.__name__.lower() + '.csv'
            filepath = os.path.join(output_dir, filename)
            instance.to_csv(
                filepath,
                update_sourceid=True,
                drop_empty_columns=drop_empty_columns,
                quiet=quiet)
```

```
if __name__ == '__main__':
    parser = argparse.ArgumentParser(
        description='A command-line script to export Specify data.')
    parser.add_argument(
        'config_file', type=argparse.FileType('rU'),
        help='path to a config-file')
    parser.add_argument(
        'output_dir', default='.', nargs='?',
        help='path to output directory')
    parser.add_argument(
        '-d', '--drop-empty-columns', dest='drop_empty_columns',
        action='store_true', help='drop columns without data')
    parser.add_argument(
        '-v', '--verbose', action='store_true')
    args = parser.parse_args()

    if not os.path.isdir(args.output_dir):
        msg = "%d is not valid directory" % args.output_dir
        raise argparse.ArgumentTypeError(msg)

    quiet = False if args.verbose else True

    export_all_data(
        args.config_file.name,
        output_dir=args.output_dir,
        drop_empty_columns=args.drop_empty_columns,
        quiet=quiet)
```

export_all_data.py

1.3.2 Importing data to multiple tables

Importing data to multiple tables in Specify is a lot more complicated than doing single table imports. The major reason is that foreign keys need to be updated before the batches of data can be loaded to the database. Another reason is that presence of self-relations in some tables (e.g. agent and taxon) may require extra updates to the database after the records have been uploaded.

Loading batches of data is faster than updating existing database records. Therefore, you can speed up imports by minimizing the number of records being updated. This can be done by importing data to tables in a preferable order. For example, it is usually good to start with the agent-table, since many other tables refer to that.

Due to contraints in the database, you are sometimes also required to create records in a certain order. For example, every record uploaded to the preparation-table must refer to an existing record in the collectionobject-table.

Tip: When building import scripts it is often a good idea to start with a single table, and then add other tables one at a time after doing some testing. You can save a dump of your database before you begin, and then have that dump restored prior to every new import trial.

For demonstration purpose, data will be uploaded to the following tables:

- agent
- locality
- collectingevent

- collector
- collectionobject
- preparationtype
- preparation

We will use some fabricated data, that in short look like this (you may recognize the agent names from the *import exercise* in the quickstart guide):

CatalogNumber	Preptype	Collector	Locality
dummy-1	herbarium sheet	Thunberg, Carl Peter	Japan
dummy-2	herbarium sheet	Thunberg, Carl Peter	Japan
dummy-3	herbarium sheet	Forsskål, Peter	Egypt
dummy-4	herbarium sheet	Forsskål, Peter	Egypt
dummy-5	herbarium sheet	Solander, Daniel	Australia
dummy-6	herbarium sheet	Solander, Daniel	Australia
dummy-7	herbarium sheet	Osbeck, Pehr	China
dummy-8	herbarium sheet	Osbeck, Pehr	China
dummy-9	herbarium sheet	Kalm, Pehr	Canada
dummy-10	herbarium sheet	Kalm, Pehr	Canada

Download a zipped archive with the data: `sample_data.zip`

The following files are included in the archive:

Filename	Specify table	# rows	# columns
agent_sample.csv	agent	5	3
locality_sample.csv	locality	5	2
collectingevent_sample.csv	collectingevent	5	2
collector_sample.csv	collector	5	4
collectionobject_sample.csv	collectionobject	10	3
preptype_sample.csv	preptype	1	2
preparation_sample.csv	preparation	10	3

The script for importing the sample data is shown below. Note that we add default values to fields that cannot be set to `NULL`.

```
#!/usr/bin/env python

"""import_sample_data.py - script for importing sample data"""

from collectionbatchtool import *

apply_user_settings('settings.cfg')  # change to your own config-file!

agt = AgentDataset()
agt.from_csv('agent_sample.csv', quiet=False)
agt.to_database(defaults={'agenttype': 1}, quiet=False)

loc = LocalityDataset()
loc.from_csv('locality_sample.csv', quiet=False)
loc.to_database(defaults={'srclatlongunit': 3}, quiet=False)

cev = CollectingeventDataset()
cev.from_csv('collectingevent_sample.csv', quiet=False)
cev.update_foreign_keys([agt, loc], quiet=False)
cev.to_database(quiet=False)
```

```
col = CollectorDataset()
col.from_csv('collector_sample.csv', quiet=False)
col.update_foreign_keys([agt, cev], quiet=False)
col.to_database(defaults={'isprimary': 1}, quiet=False)

cob = CollectionobjectDataset()
cob.from_csv('collectionobject_sample.csv', quiet=False)
cob.update_foreign_keys(cev, quiet=False)
cob.to_database(quiet=False)

pty = PreptypeDataset()
pty.from_csv('preptype_sample.csv', quiet=False)
pty.match_database_records('name')  # match existing preptypes by "name"
pty.to_database(defaults={'isloanable': 1}, quiet=False)

pre = PreparationDataset()
pre.from_csv('preparation_sample.csv', quiet=False)
pre.update_foreign_keys([pty, cob], quiet=False)
pre.to_database(quiet=False)
```

`import_sample_data.py`

1.3.3 Counting database records

You can easily count how many records there are in the database that belong to your collection. The code below counts the records in the agent-table, without the need of downloading them:

```
>>> agt = AgentDataset()
>>> agt.database_query.count()
1
```

To display the number of records for each available table, you can do something like this:

```
>>> for tabledataset_subclass in TableDataset.__subclasses__():
...     instance = tabledataset_subclass()
...     print('{0}: {1}'.format(
...         instance.model.__name__, instance.database_query.count()))
```

1.4 Preparing CSV files

One of CollectionBatchTool's underlying principles is that there should be one specified file format for each database table available for import (see which tables are available for import in the document on *supported Specify tables*). An import file should be in the comma-separated values (CSV) file format, with the first line containing column names. The names of the columns should be the same as in the corresponding database table, except for names of key-columns (see below).

1.4.1 SourceID-columns

The key-columns in import files are collectively referred to as *SourceID-columns*. These columns are functionally equivalent to the primary and foreign keys in database tables, but will not get imported directly into the database. Rather, they will be used for updating foreign key values before the data is uploaded. The names of the SourceID-columns resemble those of the database's key-fields. Here is what they look like for agent-table data:

Database key-field	SourceID-column
agentid	agent_sourceid
createdbyagentid	createdbyagent_sourceid
modifiedbyagentid	modifiedbyagent_sourceid
parentorganizationid	parentorganization_sourceid

To see the names of the key-columns, just check the *key_columns* attribute of your *TableDataset* object:

```
>>> agt = AgentDataset()
>>> agt.key_columns
{'agentid': 'agent_sourceid',
 'createdbyagentid': 'createdbyagent_sourceid',
 'modifiedbyagentid': 'modifiedbyagent_sourceid',
 'parentorganizationid': 'parentorganization_sourceid'}
```

Note: Don't know how to set up your database connetion? Have a look at the *quickstart guide*.

1.4.2 Which fields are available?

Which fields that are available for data import (or export) depends on the version of CollectionBatchTool you are using. You can get a list of available column names for the import file by examining at the *file_columns* attribute of your *TableDataset* object. The example below shows the columns available for import to (or export from) the agent-table.

```
>>> agt = AgentDataset()
>>> agt.file_columns
['agent_sourceid', 'createdbyagent_sourceid', 'modifiedbyagent_sourceid',
'parentorganization_sourceid', 'abbreviation', 'agenttype', 'dateofbirth',
'dateofbirthprecision', 'dateofdeath', 'dateofdeathprecision', 'datetype',
'email', 'firstname', 'guid', 'initials', 'interests', 'jobtitle', 'lastname',
'middleinitial', 'remarks', 'suffix', 'timestampcreated', 'timestampmodified',
'title', 'url', 'version']
```

1.5 Supported Specify Tables

This document lists the tables and *TableDataset subclasses* that are currently implemented. Only tables with a corresponding TableDataset subclass are available for batch operations (i.e. data import and export).

Specify table	TableDataset subclass	Version added
accession	AccessionDataset	0.1.4
addressofrecord	AddressofrecordDataset	0.1.4
agent	AgentDataset	
collectingevent	CollectingeventDataset	
collectingeventattribute	CollectingeventattributeDataset	0.1.2
collection	—	
collectionobject	CollectionobjectDataset	
collectionobjectattribute	CollectionobjectattributeDataset	0.1.2
collector	CollectorDataset	
determination	DeterminationDataset	
discipline	—	
division	—	
geography	GeographyDataset	
geographytreedef	—	
geographytreedefitem	GeographytreedefitemDataset	
locality	LocalityDataset	
picklist	PicklistDataset	0.1.6
picklistitem	PicklistitemDataset	0.1.6
preptype	PreptypeDataset	
preparation	PreparationDataset	
repositoryagreement	RepositoryagreementDataset	0.1.4
specifyuser	—	
storage	StorageDataset	
storagetreedef	—	
storagetreedefitem	StoragetreedefitemDataset	
taxon	TaxonDataset	
taxontreedef	—	
taxontreedefitem	TaxontreedefitemDataset	

1.6 API Reference

This document describe the API of the `collectionbatchtool` module. The following sections are included:

- *Module-level functions*
- *The TableDataset class*
- *The TreeDataset class*
- *TableDataset subclasses*

1.6.1 Module-level functions

apply_specify_context(*collection_name*, *specify_user*, *quiet=True*)
Set up the Specify context.

Parameters

- **collection_name** (*str*) – Name of an existing Specify collection.
- **specify_user** (*str*) – Username for an existing Specify user.
- **quiet** (*bool, default True*) – If True, no output will be written to standard output.

apply_user_settings (*filepath*, *quiet=True*)
Read and apply user settings in a configuration file.

Parameters

- **filepath** (*str*) – Path to the configuration file.
- **quiet** (*bool, default True*) – If True, no output will be written to standard output.

initiate_database (*database*, *host*, *user*, *passwd*, *quiet=True*)
Initiate the database.

Parameters

- **database** (*str*) – Name of a MySQL database.
- **host** (*str*) – Database host.
- **user** (*str*) – MySQL user name.
- **passwd** (*str*) – MySQL password.
- **quiet** (*bool, default True*) – If True, no output will be written to standard output.

query_to_dataframe (*database*, *query*)
Return result from a peewee `SelectQuery` as a `pandas.DataFrame`.

1.6.2 The TableDataset class

class TableDataset (*model*, *key_columns*, *static_content*, *where_clause*, *frame*)
Bases: `object`

Store a dataset corresponding to a database table.

model
`peewee.BaseModel`

A Specify data model corresponding to a table.

key_columns
dict

Key-fields and SourceID-columns for the *model*.

static_content
dict

Data to automatically inserted for the *model*.

where_clause
`peewee.Expression`

Condition for getting relevant data from the database.

describe_columns ()
Return a `pandas.DataFrame` describing the columns in the current model.

from_csv (*filepath*, *quiet=True*, ***kwargs*)
Read dataset from a CSV file.

Parameters

- **filepath** (*str*) – File path or object.
- **quiet** (*bool, default True*) – If True, no output will be written to standard output.

- ****kwargs** – Arbitrary keyword arguments available in `pandas.read_csv()`.

from_database(*quiet=True*)
Read table data from the database.

Parameters **quiet** (*bool, default True*) – If True, no output will be written to standard output.

get_match_count(*target_column*, *match_columns*)
Return counts for matches and possible matches.

Parameters

- **target_column** (*str*) – Column that should have a value if any value in *match_columns* is not null.
- **match_columns** (*str or List[str]*) – Column or columns used for updating values in *target_column*.

Returns matches, possible matches

Return type tuple

get_mismatches(*target_column*, *match_columns*)
Return a `pandas.Series` or a `pandas.DataFrame` with non-matching values.

Parameters

- **target_column** (*str*) – Column that should have a value if any value in *match_columns* is not null.
- **match_columns** (*str or List[str]*) – Column or columns used for updating values in *target_column*.

match_database_records(*match_columns*, *quiet=True*)
Update primary key values for records that match database.

Parameters

- **match_columns** (*str or List[str]*) – Columns to be matched against the database.
- **quiet** (*bool, default False*) – If True, no output will be written to standard output.

to_csv(*filepath*, *update_sourceid=False*, *drop_empty_columns=False*, *quiet=True*, *encoding='utf-8'*, *float_format='%g'*, *index=False*, ***kwargs*)
Write dataset a comma-separated values (CSV) file.

Parameters

- **filepath** (*str*) – File path or object.
- **update_sourceid** (*bool, default False*) – If True, copying ID-columns to SourceID-columns before writing to the CSV file.
- **drop_empty_columns** (*bool, default False*) – Drop columns that does not contain any data.
- **quiet** (*bool, default True*) – If True, no output will be written to standard output.
- **encoding** (*str, default 'utf-8'*) – A string representing the encoding to use in the output file.

- **float_format** (*str or None, default '%g'*) – Format string for floating point numbers.
- **index** (*bool, default False*) – Write row names (index).
- ****kwargs** – Arbitrary keyword arguments available in pandas.DataFrame.to_csv().

to_database(*defaults=None*, *update_record_metadata=True*, *chunksize=10000*, *quiet=True*)
Load a dataset into the corresponding table and update the dataset's primary key column from the database.

Parameters

- **defaults** (*dict*) – Column name and value to insert instead of nulls.
- **update_record_metadata** (*bool, default True*) – If True, record metadata will be generated during import, otherwise the metadata will be loaded from the dataset.
- **chunksize** (*int*) – Size of chunks being uploaded.
- **quiet** (*bool, default True*) – If True, no output will be written to standard output.

update_database_records(*columns*, *update_record_metadata=True*, *chunksize=10000*, *quiet=True*)
Update records in database with matching primary key values.

Parameters

- **columns** (*str or List[str]*) – Column or columns with new values.
- **update_record_metadata** (*bool, default True*) – If True, record metadata will be generated during import, otherwise the metadata will be updated from the dataset.
- **chunksize** (*int*) – Size of chunks being updated; default 1000.
- **quiet** (*bool, default True*) – If True, no output will be written to standard output.

update_foreign_keys(*from_datasets*, *quiet=False*)
Update foreign key values from a related dataset based on sourceid values.

Parameters

- **from_datasets** (*TableDataset* or List[*TableDataset*]) – Dataset(s) from which foreign key values will be updated.
- **quiet** (*bool, default False*) – If True, no output will be written to standard output.

update_sourceid(*quiet=True*)
Copy values from ID-columns to SourceID-columns.

Parameters **quiet** (*bool, default True*) – If True, no output will be written to standard output.

write_mapping_to_csv(*filepath*, *quiet=True*, *float_format='%g'*, *index=False*, ***kwargs*)
Write ID-column mapping a comma-separated values (CSV) file.

Parameters

- **filepath** (*str*) – File path or object.
- **quiet** (*bool, default True*) – If True, no output will be written to standard output.

- **float_format** (*str or None, default '%g'*) – Format string for floating point numbers.
- **index** (*bool, default False*) – Write row names (index).
- ****kwargs** – Arbitrary keyword arguments available in pandas.DataFrame.to_csv().

all_columns
List containing all columns in the dataset.

database_columns
List with available database columns.

database_query
Database query for reading the data from the database.

file_columns
List containing only the columns that can be written to or read from a file.

frame
A pandas.DataFrame to hold the data.

primary_key_column
Name of the primary key column.

1.6.3 The TreeDataset class

class TreeDataset
Bases: object

A dataset corresponding to a tree table in Specify.

update_rankid_column(*dataset*, *quiet=True*)
Update RankID based on SourceID-column.

Parameters

- **dataset** (*TableDataset*) – A treedefitem-dataset from which RankID should be updated.
- **quiet** (*bool, default True*) – If True, no output will be written to standard output.

Notes

This method exists in order to update the redundant RankID-columns in *TreeDataset* dataframes.

1.6.4 TableDataset subclasses

class AgentDataset
Bases: *collectionbatchtool.TableDataset*

Dataset corresponding to the agent-table.

class CollectingeventattributeDataset
Bases: *collectionbatchtool.TableDataset*

Dataset corresponding to the collectingeventattribute-table.

class CollectingeventDataset
Bases: *collectionbatchtool.TableDataset*

Dataset corresponding to the collectingevent-table.

class CollectionobjectattributeDataset
Bases: *collectionbatchtool.TableDataset*

Dataset corresponding to the collectionobjectattribute-table.

class CollectionobjectDataset
Bases: *collectionbatchtool.TableDataset*

Dataset corresponding to the collectionobject-table.

class CollectorDataset
Bases: *collectionbatchtool.TableDataset*

Dataset corresponding to the collector-table.

class DeterminationDataset
Bases: *collectionbatchtool.TableDataset*

Dataset corresponding to the determination-table.

class GeographyDataset
Bases: *collectionbatchtool.TableDataset*, *collectionbatchtool.TreeDataset*

Dataset corresponding to the geography-table.

class GeographytreedefitemDataset
Bases: *collectionbatchtool.TableDataset*

Dataset corresponding to the geographytreedefitem-table.

class LocalityDataset
Bases: *collectionbatchtool.TableDataset*

Dataset corresponding to the locality-table.

class StorageDataset
Bases: *collectionbatchtool.TableDataset*, *collectionbatchtool.TreeDataset*

Dataset corresponding to the storage-table.

class StoragetreedefitemDataset
Bases: *collectionbatchtool.TableDataset*

Dataset corresponding to the storagetreedefitem-table.

class PreparationDataset
Bases: *collectionbatchtool.TableDataset*

Dataset corresponding to the preparation-table.

class PreptypeDataset
Bases: *collectionbatchtool.TableDataset*

Dataset corresponding to the preptype-table.

class TaxonDataset
Bases: *collectionbatchtool.TableDataset*, *collectionbatchtool.TreeDataset*

Dataset corresponding to the taxon-table.

class `TaxontreedefitemDataset`

Bases: *collectionbatchtool.TableDataset*

Dataset corresponding to the taxontreedefitem-table.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

C

collectionbatchtool, 13

A

AgentDataset (class in collectionbatchtool), 17
all_columns (TableDataset attribute), 17
apply_specify_context() (in module collectionbatchtool), 13
apply_user_settings() (in module collectionbatchtool), 13

C

CollectingeventattributeDataset (class in collectionbatchtool), 17
CollectingeventDataset (class in collectionbatchtool), 17
collectionbatchtool (module), 13
CollectionobjectattributeDataset (class in collectionbatchtool), 18
CollectionobjectDataset (class in collectionbatchtool), 18
CollectorDataset (class in collectionbatchtool), 18

D

database_columns (TableDataset attribute), 17
database_query (TableDataset attribute), 17
describe_columns() (TableDataset method), 14
DeterminationDataset (class in collectionbatchtool), 18

F

file_columns (TableDataset attribute), 17
frame (TableDataset attribute), 17
from_csv() (TableDataset method), 14
from_database() (TableDataset method), 15

G

GeographyDataset (class in collectionbatchtool), 18
GeographytreedefitemDataset (class in collectionbatchtool), 18
get_match_count() (TableDataset method), 15
get_mismatches() (TableDataset method), 15

I

initiate_database() (in module collectionbatchtool), 14

K

key_columns (TableDataset attribute), 14

L

LocalityDataset (class in collectionbatchtool), 18

M

match_database_records() (TableDataset method), 15
model (TableDataset attribute), 14

P

PreparationDataset (class in collectionbatchtool), 18
PreptypeDataset (class in collectionbatchtool), 18
primary_key_column (TableDataset attribute), 17

Q

query_to_dataframe() (in module collectionbatchtool), 14

S

static_content (TableDataset attribute), 14
StorageDataset (class in collectionbatchtool), 18
StoragetreedefitemDataset (class in collectionbatchtool), 18

T

TableDataset (class in collectionbatchtool), 14
TaxonDataset (class in collectionbatchtool), 18
TaxontreedefitemDataset (class in collectionbatchtool), 18
to_csv() (TableDataset method), 15
to_database() (TableDataset method), 16
TreeDataset (class in collectionbatchtool), 17

U

update_database_records() (TableDataset method), 16
update_foreign_keys() (TableDataset method), 16
update_rankid_column() (TreeDataset method), 17
update_sourceid() (TableDataset method), 16

W

where_clause (TableDataset attribute), 14
write_mapping_to_csv() (TableDataset method), 16