Pyrseas Documentation

Release 0.9.0

Joe Abbate

Apr 14, 2020

Contents

Pyrseas provides utilities to describe a PostgreSQL database schema as YAML, to verify the schema against the same or a different database and to generate SQL that will modify the schema to match the YAML description.

CHAPTER 1

Features

- Outputs a YAML/JSON description of a PostgreSQL database's tables and other objects (metadata), suitable for storing in a version control repository
- Generates SQL statements to modify a database so that it willl match an input YAML/JSON specification
- Generates an augmented YAML description of a PostgreSQL database from its catalogs and an augmentation specification.

Requirements

- PostgreSQL 9.4 or higher
- Python 2.7, 3.4 or higher
- Psycopg2 2.7 or higher
- PyYAML 3.13 or higher
- PgDbConn 0.8 or higher

Contents

3.1 Overview

Pyrseas provides utilities to maintain a PostgreSQL database schema. Its purpose is to enhance and follow through on the concepts of the Andromeda Project.

Whereas Andromeda expects the database designer or developer to provide a single YAML specification file of the database to be created, Pyrseas allows the development database to be created using the familiar SQL CREATE statements. The developer can then run the *dbtoyaml* utility to generate the YAML specification from the database. The spec can then be stored in any desired version control (VCS) repository. Similarly, she can add columns or modify tables or other objects using SQL ALTER statements and regenerate the YAML spec with dbtoyaml.

When ready to create or upgrade a test or production database, the *yamltodb* utility can be used with the YAML spec as input, to generate a script of SQL CREATE or ALTER statements to modify the database so that it matches the input spec.

A third tool, *dbaugment*, can be used to add custom attributes and supporting objects to a given schema. For example, an *updated* column can be added to various tables, together with trigger functions to ensure the columns are automatically modified as changes are made.

3.1.1 Use Cases

The following sections discuss the main scenarios where Pyrseas tools may be helpful.

3.1.2 Version Control

The case for implementing a tool to facilitate version control over SQL databases was made in a couple of blog posts: Version Control, Part 1: Pre-SQL and Version Control, Part 2: SQL Databases. In summary, SQL data definition commands are generally incompatible with traditional version control approaches which usually require comparisons (diffs) between revisions of source files.

A refinement of the approach described in the aforementioned blog posts will be of interest to users with many objects in their database schemas, i.e., many tables, views, functions, and other more complex objects. Instead of storing a

complete database specification in a single YAML file, by using the *–multiple-files* option to **dbtoyaml**, the specification can be broken down into files corresponding, generally, to a single database object. This allows a VCS **diff** facility to easily highlight database changes. Please refer to the *dbtoyaml - Database to YAML* and *yamltodb - YAML to Database* utilities for further details.

The Pyrseas version control tools are not designed to be the ultimate SQL database version control solution. Instead, they are aimed at assisting two or more developers or DBAs in sharing changes to the underlying database as they implement a database application. The sharing can occur through a distributed or centralized VCS. The Pyrseas tools may even be used by a single DBA in conjunction with a distributed VCS to quickly explore alternative designs. The tools can also help to share changes with a conventional QA team, but may require additional controls for final releases and production installations.

3.1.3 Supplementary Actions

In many instances, a database schema needs to be supplemented by rarely-modified data kept in certain tables, e.g., a codes-descriptions table. The data import and export features, controlled by *datacopy* configuration parameter (see *Configuration Items* for details) facilitates this need.

In other cases, DBAs may want to standardize certain additional table columns or processing. For example, they may want to capture the user and time of modification of a certain set of tables using a common procedure. The *dbaugment - Augment a database* utility was introduced to support these needs.

Generating SQL by determining what changed between one schema version and another is sometimes not sufficient. Although the change may be as simple as adding a column to a table and adding a referential constraint to the new column, if the tables already have data, it may not be possible to run the SQL generated by **yamltodb**. This requires manual intervention by the DBAs or developers. The project would like to assist with these types of changes. The blog post The Future of Pyrseas: Part 1 is a first step in discussing this requirement.

3.1.4 Naming

The project name comes from Python, the programming language, and Perseas[1], the Greek mythological hero who rescued Andromeda from a sea monster[2]. It is hoped that Pyrseas will rescue the Andromeda project <grin>. You can pronounce Pyrseas like the hero.

3.2 Installation

3.2.1 Summary

For the latest release, use:

```
pip install Pyrseas
```

For development:

```
git clone git://github.com/perseas/Pyrseas.git
cd Pyrseas
python setup.py install
```

---

[1] The common English name for Perseas is Perseus and the Ancient Greek name is Perseos. However, in modern Greek $\Pi\epsilon\rho\sigma\alpha$ is the more common spelling for the mythical hero.

[2] He is better known for having killed Medusa.

3.2.2 Requirements

Pyrseas provides tools for Postgres, so obviously you need **Postgres** to start with. Pyrseas has been tested with PG 9.4, 9.5, 9.6, 10 and 11, and we'll certainly keep up with future releases. Please refer to the Postgres download page to find a distribution for the various Linux, Unix and Windows platforms supported.

You will also need **Python**. Pyrseas was originally developed using Python 2 and been tested with Python 2.7, but that version will no longer be supported effective 1 Jan 2020. It has been ported to Python 3 and tested against versions from 3.2 through 3.7. Python 3 is the preferred usage and development environment. On Linux or *BSD, Python may already be part of your distribution or may be available as a package. For Windows and Mac OS please refer to the Python download page for installers and instructions.

Pyrseas talks to the Postgres DBMS via the **Psycopg2 adapter**. Pyrseas has been tested with psycopg2 2.5 through 2.7. Psycopg2 is available as a package on most Linux or *BSD distributions and can also be downloaded or installed from PyPI. Please refer to the Psycopg download page for more details.

Note: If you install Pyrseas using `pip` (see below) and you have not already installed Psycopg2, e.g., when installing into a `virtualenv` environment created with `--no-site-packages`, you may need to have installed the Postgres and Python development packages, and a C compiler, as `pip` may download and attempt to build and install psycopg2 before installing Pyrseas.

The Pyrseas utilities rely on **PyYAML**, a YAML library. This may be available as a package for your operating system or it can be downloaded from the Python Package Index (PyPI).

The utilities also rely on **PgDbConn**, an offshoot of the project that generalizes the Postgres database connection code used in the utilities. It can be downloaded from PyPI, but should be installed automatically when using `pip`.

3.2.3 Downloading

Pyrseas is available at the following locations:

- Python Package Index
- Postgres Extension Network (PGXN)
- GitHub repository

You can download the distribution from PyPI in gzip-compressed tar or ZIP archive format, but you can download *and* install it using `Pip`. See *Python Installer* below for details.

PGXN provides a ZIP archive which you can download or you can download *and* install using the PGXN client (see *PGXN Client* below).

The GitHub repository holds the Pyrseas source code, tagged according to the various releases, e.g., v0.9.0, and including unreleased modifications. To access it, you need Git which is available as a package in most OS distributions or can be downloaded from the Git download page. You can fetch the Pyrseas sources by issuing one of the following commands:

```
git clone git://github.com/perseas/Pyrseas.git
```

or:

```
git clone https://github.com/perseas/Pyrseas.git
```

This will create a `Pyrseas` directory tree (you can use a different target name by adding it to the above commands). To list available releases, change to the subdirectory and invoke `git tag`. To switch to a particular release, use:

```
git checkout vn.n.n
```

where *vn.n.n* is the release identifier. Use `git checkout master` to revert to the main (master) branch. To fetch the latest updates, use:

```
git pull
```

3.2.4 Installation

Extracting Sources

Once you have downloaded an archive from PyPI or PGXN, you need to extract the sources. For a gzip-compressed tar file, use:

```
tar xzf Pyrseas-n.n.n.tar.gz
```

where *n.n.n* is the release version. For a ZIP archive, use:

```
unzip Pyrseas-n.n.n.zip
```

Both commands above will create a directory `Pyrseas-n.n.n` and you will want to `cd` to it before proceeding with the installation.

Installing

If you have superuser or similar administrative privileges, you can install Pyrseas for access by multiple users on your system. On Linux and other Unix-flavored systems, you can install from the extracted `Pyrseas-n.n.n` source directory or from the root directory of the `git` clone, using the following command:

```
sudo python setup.py install
```

That will install the *dbtoyaml* and *yamltodb* utility scripts in a directory such as `/usr/local/bin`. The library sources and bytecode files will be placed in a `pyrseas` subdirectory under `site-packages` or `dist-packages`, e.g., `/usr/local/lib/python3.6/dist-packages/pyrseas`.

On Windows, from an account with Administrator privileges, you can use:

```
python setup.py install
```

That will install the Pyrseas utilities in the `Scripts` folder of your Python installation. The source and bytecode files will go in the `site-packages` folder, e.g., `C:\Python36\Lib\site-packages\pyrseas`.

Python Installer

You can also download and install Pyrseas using pip. For example, on Linux do:

```
sudo pip install Pyrseas
```

If this is the first time you are installing a Python package, please do yourself a favor and read and follow the instructions in the "Distribute & Pip" subsection of the "Installing Python on . . . " section for your platform of the The Hitchhiker's Guide to Python!.

Note: On FreeBSD, it has been reported that it is necessary to install the Python `distribute` package, prior to installing Pyrseas with `pip`. This may also be necessary on other BSD variants.

Note: On Windows 64-bit, it has been reported that it is necessary to obtain unofficial versions of the `distribute` and `PyYAML` packages, available at University of California, Irvine. For a detailed tutorial, see this post.

`Pip` can also be used in a Python virtualenv environment, in which case you *don't* need to prefix the commands with `sudo`.

`Pip` also provides the ability to uninstall Pyrseas.

PGXN Client

The PGXN client (available at PyPI) can be used to download and install Pyrseas from PGXN. Usage is:

```
pgxn install pyrseas
```

3.3 Configuration

The Pyrseas utilities allow you to configure various options through a number of YAML specification files, none of which are required–but the system configuration file is provided by the normal installation.

If a configuration parameter is specified in more than one file, the latter file in the list of files below overrides any earlier specification. Any configuration item specified on the command line takes precedence over any such item in a configuration file.

3.3.1 Configuration File Name

The default configuration file name is `config.yaml`. If desired, you can override this with the environment variable `PYRSEAS_CONFIG_FILE`, but be aware that this will affect all three levels below.

3.3.2 System Configuration

The system configuration file is distributed with Pyrseas and is normally installed in the `pyrseas` library directory.

If desired, you can override this using the `PYRSEAS_SYS_CONFIG` environment variable. This can be defined as a full path, including a file name, or a directory location, in which case the default file name as mentioned above under *Configuration File Name* will be appended to the path.

Currently, this file includes specifications for functions, triggers and other objects used by the **dbaugment** utility. It also includes the default directory path for storing multiple YAML files in a VCS repository, and the path to data files for use by the data import and export facilities.

3.3.3 User Configuration

Each user can have his or her own configuration file. The default location for this depends on the platform. Under Linux, BSD, OS/X and other Unix variants, place the file under your home directory, in the subdirectory `.config/pyrseas/`. Under Windows, put the file in `%APPDATA%\pyrseas\`.

You can override the location of the user configuration file using the PYRSEAS_USER_CONFIG environment variable. This can be defined as a full path, including a file name, or a directory location, in which case the default file name as mentioned above under *Configuration File Name* will be appended to the path.

If present, the user configuration file will be merged with the system configuration.

It is recommended that the user configuration file only be used for non-project-specific purposes. For example, if you frequently use Pyrseas against a remote database or on a non-standard port, you can specify the host or port in your personal configuration file.

3.3.4 Repository Configuration

A configuration file can be placed in a version control repository or project directory, so that it can be under version control together with other Pyrseas files such as the output from dbtoyaml --multiple-files. The default location for the repository can be specified in the user configuration, using the keys repository and path, for example:

```
repository:
  path: /home/user/project/repo
```

You can also use the --repository command line option to specify (or override) the directory path to the root of the repository and the utilities will look for a configuration file in that location.

If present, the repository configuration file will be merged with the system and user configuration information.

3.3.5 Command Line Configuration

The utilities also allow you to specify a fourth configuration file on the command line, using the --config command line option. Again, if the file exists, its information will be merged with previously read files.

3.4 Configuration Items

The following lists the various sections allowed in a configuration file and the items that are recognized by the Pyrseas utilities.

3.4.1 Augmenter

This section is used by the **dbaugment** utility (see *dbaugment - Augment a database*). Most of these are specified in the system configuration file delivered with Pyrseas, but can also be included or overriden in user or repository configuration files.

- audit_columns: This section defines combinations of columns and triggers to be added to tables. Both columns and triggers are specified as YAML lists (to be consistent with **dbtoyaml** YAML output), although normally a single trigger will be necessary per column combination. The columns and triggers should reference previously defined items in the columns and triggers sections (see below). See *Predefined Database Augmentations* for audit columns defined in the system config.yaml.

- columns: This section defines prototype columns to be added to a table by Augmenter. For each column, a valid Postgres data type should be included.

  You can also add a not_null constraint and a default specification. See *Predefined Database Augmentations* for columns defined in the system config.yaml. In a repository or user configuration file, you can also specify an alternate name for a previously defined column. For example, if you prefer that the

modified_timestamp columns be named last_update, you can add the following to a configuration file:

```
augmenter:
  columns:
    modified_timestamp:
      name: last_update
```

- function_templates: This section defines the source text for the trigger functions (see below) using a template language. Any text enclosed in double braces, e.g., {{modified_by_user}}, will be replaced, typically by a previously defined column or its alternate name (see above).

- functions: This section defines prototype trigger functions to be invoked by audit columns or other augmentations. The following items can be specified for each function:

    - description: Text for a COMMENT statement on the function.

    - language: Procedural language, e.g., plpgsql, in which the function is written.

    - returns: Value should be trigger.

    - security_definer: Indicates whether the function is to be executed with the privileges of the user that created it. This is usually needed for audit column trigger functions.

    - source: This is usually a reference to a function template (see above) enclosed in double braces, e.g., {{functempl_audit_default}}. However, in user or repository configurations, this can also be the actual text of the function.

    See *Predefined Database Augmentations* for functions defined in the system configuration file.

- schema pyrseas: This section currently defines three functions that may be installed in the pyrseas schema if the full audit columns specifications is added for Augmenter processing.

- schemas and tables: Multiple schema schema-name sections can be present, typically in a repository configuration file. Each such section can include table table-name items, and under each the audit_columns specifications to be added to the given table. For example:

```
augmenter:
  schema public:
    table t1:
      audit_columns: default
```

- triggers: This section defines the prototype triggers to be used with audit columns and other augmentations. The following items can be specified for each trigger:

    - events: This is a list that can include one or more of insert, update or delete (the latter is not used for audit columns but may be used in future augmentations).

    - level: This can take the values row or statement (usually the former).

    - name: This specifies the name to be given to a trigger. It can be a template using {{table_name}} which will then be replaced with the actual table name on which the trigger will act.

    - procedure: This is the invocation name, e.g., audit_default() of the function to be called when the trigger fires.

    - timing: This can take the values before or after (usually the former).

3.4.2 Database

This section is primarily for a user configuration file. If you frequently connect to a particular host, port or as a given user, that are *not* the Postgres defaults, adding corresponding entries to your user configuration file allows you to automatically override the defaults. If for a given invocation you need to connect to or as a different host, port or user, you can still override the configuration using the command line options (see *Common Command Line Options*):

- host: Name of the host to connect. Please refer to the Postgres connection host documentation for details and defaults.

- port: Port number to connect to. See the Postgres connection port documentation for more.

- username: Name of the user to connect as. View the Postgres connection user documentation for more.

3.4.3 Datacopy

This section is normally in a user or repository configuration file. It is used by **dbtoyaml** and **yamltodb** to determine which tables should be exported from or imported to the database. It consists of schema names, using the format *schema schema_name*, followed by lists of table names. For example:

```
datacopy:
  schema public:
  - t1
  - t2
  schema s1:
  - t3
```

3.4.4 Repository

This section is used by all utilities (but **dbaugment** does not fully support it). The "repository" is intended to be a version control, e.g., Git, Mercurial, or Subversion, repository.

- data: Path, relative to the root of the repository, where **dbtoyaml** and **yamltodb** place or expect the files containing data exported from or imported to the database. The tables to be exported or imported are specified in the `Datacopy` section. The default value (defined in the system `config.yaml`) is **metadata**.

- metadata: Path, relative to the root of the repository, where **dbtoyaml** and **yamltodb** place or expect the YAML specification files for the database objects when the *–multiple-files* option is used. The default value (defined in the system `config.yaml`) is **metadata**.

- path: Absolute path to the root of the repository. This should normally be specified in a user configuration file, or in a file given with the `--config` option. If not specified, this defaults to the current working directory from which the utility is run.

3.5 Development

The following details the tools needed to contribute to the development of Pyrseas. If you have any doubts or questions, please open an issue on GitHub (https://github.com/perseas/Pyrseas/issues). In addition, see *Version Control* below on how to set up a GitHub account to participate in development.

3.5.1 Requirements

- Git

- Python

- Postgres

- Psycopg2

- PyYAML

- PgDbConn

- Tox

3.5.2 Version Control

Pyrseas uses Git to control changes to its source code. As mentioned under *Downloading*, the master Git repository is located at GitHub.

To install Git, either download and install the latest stable release for your platform or follow the *Pro Git* installation instructions. For most Linux users, `apt-get` or `yum` (depending on Linux flavor) will be the simplest means to install the `git-core` package. For Windows, downloading the installer and selecting `Git Bash` gives you not only Git but a Bash shell, which is handy if you're coming from a Linux/Unix background.

Once Git is installed, change to a suitable directory and clone the master repository:

```
git clone git://github.com/perseas/Pyrseas.git
```

or:

```
git clone https://github.com/perseas/Pyrseas.git
```

To be able to create a fork on GitHub, open an issue or participate in Pyrseas development, you'll first have to create a GitHub account.

3.5.3 Programming Language

To contribute to Pyrseas, you need at least one version of Python. You can develop using Python 3, but since we will continue supporting Python 2 until its end-of-life, you can also install Python 2.7 in addition to Python 3.7 or 3.6.

If Python is not already available on your machine, either download and install one or both of the production releases for your platform, follow the applicable installation instructions given in The Hitchhiker's Guide to Python! or install it from your platform's package management system.

3.5.4 Database Installation

To participate in Pyrseas development, you'll also need one or more installations of Postgres, versions 11, 10, 9.6, 9.5 or 9.4. If you only have limited space, it is preferable to install one of the latest two versions.

The versions can be obtained as binary packages or installers from the Postgres.org website. The site also includes instructions for installing from package management systems or building it from source.

To access Postgres from Python, you have to install the Psycopg adapter. You can either follow the instructions in Psycopg's site, or install it from your package management system. Note that if you install both Python 2 and 3, you will have to install two packages, e.g., `python-psycopg2` and `python3-psycopg2`.

3.5.5 Other Libraries and Tools

The `dbtoyaml` and `yamltodb` utilities use the PyYAML library. You can install it from the PyYAML site, or possibly from your package management system. For Windows 64-bit, please read the note under *Python Installer*.

The utilities also rely on PgDbConn, an offshoot of the Perseas project that provides a thin, object-oriented layer over Psycopg2. You can install it from PyPI.

To easily run the Pyrseas tests against various Python/Postgres version combinations, you will need pytest and Tox. Please refer to *Testing* for more information.

3.6 Testing

The majority of Pyrseas' capabilities are exercised and verified via unit tests written using pytest. The tests can be run from the command line by most users, e.g.,

```
py.test tests/dbobject/test_table.py
py.test tests/dbobject/test_trigger.py -k test_create_trigger
py.test tests/functional
```

The first `python` command above runs all tests related to tables, mapping, creating, dropping, etc. The second one executes a single test to generate SQL to create a trigger. The third runs all the functional tests. Please review the pytest documentation for further options.

3.6.1 Environment Variables

By default, the tests use a Postgres database named `pyrseas_testdb` which is created if it doesn't already exist. The tests are run as the logged in user, using the `USER` Unix/Linux environment variable (or `USERNAME` under Windows). They access Postgres on the local host using the default port number (5432).

The following four environment variables can be used to change the defaults described above:

- PYRSEAS_TEST_DB
- PYRSEAS_TEST_USER
- PYRSEAS_TEST_HOST
- PYRSEAS_TEST_USER

3.6.2 Restrictions

Unless the test database exists and the user running the tests has access to it, the user role will need CREATEDB privilege.

Most tests do not require special privileges. However, certain tests may require Postgres SUPERUSER privilege. Such tests will normally be skipped if the user lacks the privilege.

Most tests do not require installation of supporting Postgres packages. However, a few tests rely on the availability of Postgres `contrib` modules such as the spi module or procedural languages such as `plperl` or `plpython3u`.

On Windows, it is necessary to install Perl in order to run some of the tests (most Linux or Unix variants already include it as part of their normal distribution). The last time we checked, a suitable choice appeared to be Strawberry Perl which can be downloaded from http://strawberryperl.com/releases.html. However, the default installation is placed in `C:\strawberry` and can hold a single Perl version. Furthermore, some Postgres versions may be linked with

non-current Perl versions. It is recommended that the latest Perl version be installed as this will usually give the fewest test failures. See this blog post for more details.

The COLLATION tests require the `fr_FR.utf8` locale (or `French.France.1252` language on Windows) to be installed.

3.6.3 Testing Checklist

The following is a summary list of steps needed to test Pyrseas on a new machine. Refer to *Development* for details on how to accomplish a given installation task. "Package manager" refers to the platform's package management system utility such as `apt-get` or `yum`. Installation from PyPI can be done with `pip`. Some operations require administrative or superuser privileges, at either the operating system or Postgres level.

- Install Git using package manager or from https://git-scm.com/download (on Windows, prefer Git Bash)

- `git clone git://github.com/perseas/Pyrseas.git`

- Install Python 3.7 (or 3.6) and 2.7, using package manager or from installers at https://www.python. org/downloads/.

- Install Postgres 11, 10, 9.6, 9.5 and 9.4, using package manager or binary installers at https://www. postgresql.org/download/

  Note: On Linux, make sure you install the contrib and plperl packages, e.g., on Debian, postgresql-contrib-n.n and postgresql-plperl-n.n (where *n.n* is the Postgres version number–or simply *n* from Postgres 10 onward)

- Install Psycopg2, using package manager, or from PyPI (https://pypi.org/project/psycopg2/) or http://initd.org/psycopg/download/.

  Note: On Windows, you may first want to install a version of Microsoft Visual Studio from here. An alternative that may work is MinGW. See these blog posts for more details.

- Install PyYAML, using package manager, or from PyPI (https://pypi.org/project/PyYAML/) or http://pyyaml.org/download/pyyaml/.

- Install PgDbConn from PyPI (https://pypi.org/project/pgdbconn/).

- Install pytest, using package manager, or from PyPI (https://pypi.org/project/pytest/).

- Install Tox, using package manager, or from PyPI (https://pypi.org/project/tox/)

  Note: Psycopg2, PyYAML, pytest and Tox all have to be installed twice, i.e., once under Python 3.7 (or 3.6) and another under 2.7.

- On Windows, install Perl (see discussion above under "Restrictions"). On Linux, usually Perl is already available.

- As **postgres** user, using psql or pgAdmin, create a test user, e.g., your name. The user running tests must have at a minimum createdb privilege, in order to create the test database. To run *all* the tests, the user also needs superuser privilege.

- Create a Postgres password file, e.g., on Linux: `~/.pgpass`, on Windows: `%APPDATA%\postgresql\pgpass.conf`.

- Create directories to hold tablespaces, e.g., `/extra/pg/11.0/ts1` on Linux, `C:\\extra\\pg\\11.0\\ts1` on Windows. The directories need to be owned by the **postgres** user. This may be tricky on older Windows versions, but the command `cacls <dir> /E /G postgres:F` should suffice. Using `psql`, create tablespaces **ts1** and **ts2**, e.g., `CREATE TABLESPACE ts1 LOCATION '<directory>'` (on Windows, you'll have to use, e.g., `E'C:\\dir\\ts1'`, to specify the directory).

- Install the locale `fr_FR.utf8` on Linux/Unix or the language `French.France.1252` on Windows.

    - On Debian and derivatives, this can be done with the command:

      ```
      sudo dpkg-reconfigure locales
      ```

    - On Windows, open the Control Panel, select Date, Time, Language, and Regional Options, then Regional and Language Options (or Add other languages), click on the Advanced tab in the dialog and then choose "French (France)" from the dropdown. Finally, click OK and respond to any subsequent prompts to install the locale, including rebooting the machine.

- Change to the Pyrseas source directory (created by the second step above).

    - Define the `PYTHONPATH` environment variable to the Pyrseas source directory, e.g., on Linux, `export PYTHONPATH=$PWD`, on Windows, `set PYTHONPATH=%USERPROFILE%\somedir\Pyrseas`.

    - Define the environment variables `PG94_PORT`, `PG95_PORT`, `PG96_PORT`, `PG100_PORT` and `PG110_PORT` to point to the corresponding Postgres connection ports.

- Invoke `tox`. This will create two virtualenvs in a `.tox` subdirectory–one for Python 3.7 or 3.6 and another for 2.7, install Pyrseas and its prerequisites (Psycopg2 and PyYAML) into each virtualenv and run the unit tests for each combination of Postgres and Python.

If you find any problems with the instructions above, please open an issue on GitHub.

3.7 Known Issues

The following summarizes notable deficiencies in the current release of the Pyrseas utilities. For further details please refer to the discussions in the Pyrseas issue tracker. Suggestions or patches to deal with these issues are welcome.

3.7.1 Coverage of Postgres Objects

An important Pyrseas objective is to support creating, altering or dropping nearly any Postgres object accessible through SQL, including adding, modifying or removing any attributes or features of those objects. At present, we believe Pyrseas covers roughly over 90% of the Postgres object/attribute universe. Please refer to the Feature Matrix for details.

This is a continuing effort since Postgres keeps adding new features in each release, such as the table PARTITIONING syntax in PG 10. We have documented current limitations in the issue tracker, see, for example, issues 135 and 178. Please open an issue on the tracker if you find objects or features needing additional support.

3.7.2 Object Dependencies

The first releases of **yamltodb** used a generally fixed traversal order when generating SQL. This caused problems with complex dependencies between objects (e.g., views that depended on functions that depended on types). Release 0.8 introduced a topological sort of objects based on their dependencies. The resulting dependency graph is now used

to drive SQL generation. This should eliminate most object dependency problems seen with the previous architecture. However, certain issues still remain. Specifically, if an object depends on a Postgres internally-defined object, or on an object defined by a Postgres extension, the Pyrseas utilities may not behave as expected (see issue 175 for additional discussion).

3.7.3 Object renaming

Pyrseas provides support for generating SQL statements to rename various database objects, e.g., ALTER TABLE t1 RENAME TO t2, using an 'oldname' tag which can be added to objects that support SQL RENAME. The tag has to be added manually to a YAML specification for yamltodb to act on it and cannot be kept in the YAML file for subsequent runs. This is not entirely satisfactory for storing the YAML file in a version control system.

3.7.4 Memory utilization

The yamltodb utility compares the existing and input metadata by constructing parallel, in-memory representations of the database catalogs and the input YAML specification. If the database has a large number of objects, e.g., in the thousands of tables, the utility's memory usage may be noticeable.

3.7.5 Multiline Strings

The text of function source code, view definitions or object COMMENTs present a problem when they span multiple lines. The default YAML output format is to enclose the entire string in double quotes, to show newlines that are part of the text as escaped characters (i.e., \n) and to break the text into lines with a backslash-newline-indentation-backslash pattern. For example:

```
source: "\n    SELECT inventory_id\n    FROM inventory\n    WHERE film_id =\
 \ $1\n    AND store_id = $2\n    AND inventory_in_stock(inventory_id);\n"
```

This is not very readable, but it does allow YAML to read it back and correctly reconstruct the original string. To improve readability, Pyrseas 0.7 introduced special processing for these strings. By using YAML notation, the same string is represented as follows:

```
source: |2

    SELECT inventory_id
    FROM inventory
    WHERE film_id = $1
    AND store_id = $2
    AND NOT inventory_in_stock(inventory_id);
```

However, due to Python 2.x issues with Unicode, the more readable format is *only* available if using Python 3.x.

Note also that if your function source code has trailing spaces at the end of lines, they would normally be represented in the original default format. However, in the interest of readability, **dbtoyaml** will remove the trailing spaces from the text.

3.7.6 Index and Partitioning Expressions

Postgres allows users to create indexes using expressions. A user can also mix expressions with regular columns. The Postgres catalogs store the index information in a bespoke fashion: an array of column numbers where a zero indicates an expression and a list of expression trees (an internal format) for the expressions, with additional arrays for collation information, operator classes and index options such as ASC or DESC. Although the pg_get_indexdef system

catalog function can be used to obtain a full CREATE INDEX statement, Pyrseas has chosen to specify each column or expresssion separately in the YAML definitions. This has not been satisfactory in complex cases (see for example issue 170) and is an area requiring further attention. A similar situation exists for table partitioning using expresssions.

3.8 Predefined Database Augmentations

These augmentations are specified in the config.yaml configuration file distributed with Pyrseas' **dbaugment**.

3.8.1 Columns

These are predefined column specifications that can be added to tables, e.g., in various audit column combinations (see *Audit Columns* below).

- created_by_ip_address: An INET column to record the IP address which originated the current row.
- created_by_user: A VARCHAR(63) column to record the user, e.g., CURRENT_USER, who created the current row.
- created_date: A DATE column that defaults to CURRENT_DATE.
- created_timestamp: A TIMESTAMP WITH TIME ZONE column to record the date and time when the current row was created.
- modified_by_ip_address: An INET column to record the IP address which originated the last modification to the current row.
- modified_by_user: A VARCHAR(63) column to record the user, e.g., CURRENT_USER, who last modified the current row.
- modified_timestamp: A TIMESTAMP WITH TIME ZONE column to record the date and time when the current row was last modified.

3.8.2 Functions

The following are predefined trigger functions which are used to implement various augmentations. The source for each function, written in PL/pgSQL, is specified in a function template, named with a functempl_ prefixed to the function name.

- Audit when modified (audit_modified): This function provides the CURRENT_TIMESTAMP value for audit columns.
- Default audit (audit_default): This function provides the CURRENT_USER and CURRENT_TIMESTAMP for audit columns.
- Full audit (audit_full): For SQL INSERTs, this function provides values for the user who created the row, the CURRENT_TIMESTAMP and the IP address for both the created_ and modified_ audit columns. For UPDATEs, it retains the existing values in the created_ columns and supplies current values for the modified_ columns.

In addition, the following helper functions are defined in schema pyrseas:

- get_session_variable
- set_session_variable

A variant of `get_session_variable` is invoked by the `audit_full` function to retrieve the actual (logged-on) user and IP address. In web applications, the user that connects to the database is typically the system user running the web server, rather than the application (logged on) user. The application can invoke the `pyrseas.set_session_variable` function to supply the application user and IP address so that the audit trail will reflect the application context corrrectly.

3.8.3 Audit Columns

These are predefined combinations of columns to be added to tables to record audit trail information. They may also include triggers to be invoked to maintain the column values.

- created_date_only: This is the simplest audit trail that adds a `created_date` column which defaults to the CURRENT_DATE.

- modified_only: This is another simple audit trail. It adds a `modified_timestamp` column which is supplied by a trigger named *table_name*_20_audit_modified_only.

- default: This is the default for audit columns. It adds the columns `modified_by_user` and `modified_timestamp` and a trigger named *table_name*_20_audit_default to fill in the columns.

- full: This is the most extensive audit trail combination. It adds `created_` and `modified_` columns for user, IP address and timestamp. It also adds a trigger named *table_name*_20_audit_full.

3.9 dbaugment - Augment a database

3.9.1 Name

dbaugment – Augment a Postgres database in predefined ways

3.9.2 Synopsys

```
dbaugment [option...] dbname [spec]
```

3.9.3 Description

dbaugment is a utility for augmenting a Postgres database with various standard attributes and procedures, such as automatically maintained audit columns. The augmentations are defined in a YAML-formatted `spec` file.

The following is an example of a specification file:

```
augmenter:
  columns:
    modified_date:
      not_null: true
      type: date
schema public:
  table t1:
    audit_columns: default
  table t3:
    audit_columns: modified_only
```

The specification file lists each schema, and within it, each table to be augmented. Under each table the following values are currently recognized:

- audit_columns: This indicates that audit trail columns are to be added to the table, e.g., a timestamp column recording when a row was last modified.

The first section of the specification file, under the `augmenter` header, lists configuration information. This is in addition to the built-in configuration objects (see *Predefined Database Augmentations*).

dbaugment first reads the database catalogs. It also initializes itself from predefined configuration information. **dbaugment** then reads the specification file, which may include additional configuration objects, and outputs a YAML file, including the existing catalog information together with the desired enhancements. The YAML file is suitable for input to **yamltodb** to generate the SQL statements to implement the changes.

3.9.4 Options

dbaugment accepts the following command-line arguments (in addition to the *Common Command Line Options*):

dbname

Specifies the name of the database whose schema is to augmented.

spec

Location of the file with the augmenter specifications. If this is omitted, the specification is read from the program's standard input.

3.9.5 Examples

To augment a database called `moviesdb` according to the specifications in the file `movies.yaml`:

```
dbaugment moviesdb movies.yaml
```

To add a column named `updated` to table `public.film` to hold the date and time each row was inserted or updated, create a YAML specification file, say `film.yaml` as follows:

```
augmenter:
  columns:
    modified_timestamp:
      name: updated
schema public:
  table film:
    audit_columns: modified_only
```

The first four lines configure the predefined `modified_timestamp` audit column to use the name `updated` instead. The last three lines direct `dbaugment` to apply the predefined `modified_only` audit column to the `film` table.

Then run the following command to generate the resulting database specification, alter the table and create the needed trigger and function.

dbaugment moviesdb film.yaml | yamltodb moviesdb -u

3.9.6 See Also

Predefined Database Augmentations

3.10 dbtoyaml - Database to YAML

3.10.1 Name

dbtoyaml – extract the schema of a Postgres database in YAML format

3.10.2 Synopsys

```
dbtoyaml [option...] dbname
```

3.10.3 Description

dbtoyaml is a utility for extracting the schema of a Postgres database to a YAML formatted specification. By default, the specification is output as a single output stream, which can be redirected or explicitly sent to a file. As an alternative, the `--multiple-files` option allows you to break down the specification into multiple files, in general, one for each object (see *Multiple File Output*).

Note that JSON is an official subset of YAML version 1.2, so the **dbtoyaml** output should also be compatible with JSON tools.

A sample of the output format is as follows:

```
schema public:
  owner: postgres
  privileges:
  - postgres:
    - all
  - PUBLIC:
    - all
  table t1:
    check_constraints:
      t1_c2_check:
        columns:
        - c2
        expression: (c2 > 123)
    columns:
    - c1:
        not_null: true
        type: integer
    - c2:
        type: smallint
    - c3:
        default: 'false'
        type: boolean
    - c4:
        type: text
    foreign_keys:
      t1_c2_fkey:
        columns:
        - c2
        references:
          columns:
          - c21
          schema: s1
```

(continues on next page)

```
        table: t2
    owner: alice
    primary_key:
      t1_pkey:
        columns:
        - c1
schema s1:
  owner: bob
  privileges:
  - bob:
    - all
  - alice:
    - all
  table t2:
    columns:
    - c21:
        not_null: true
        type: integer
    - c22:
        type: character varying(16)
    owner: bob
    primary_key:
      t2_pkey:
        columns:
        - c21
    privileges:
    - bob:
      - all
    - PUBLIC:
      - select
    - alice:
      - insert:
          grantable: true
      - delete:
          grantable: true
      - update:
          grantable: true
    - carol:
        grantor: alice
        privs:
        - insert
```

The above should be mostly self-explanatory. The example database has two tables, named `t1` and `t2`, the first –owned by user 'alice'– in the `public` schema and the second –owned by user 'bob'– in a schema named `s1` (also owned by 'bob'). The `columns:` specifications directly under each table list each column in that table, in the same order as shown by Postgres. The specifications `primary_key:`, `foreign_keys:` and `check_constraints:` define PRIMARY KEY, FOREIGN KEY and CHECK constraints for a given table. Additional specifications (not shown) define unique constraints and indexes.

User 'bob' has granted all privileges to 'alice' on the `s1` schema. On table `t2`, he also granted SELECT to PUBLIC; INSERT, UPDATE and DELETE to 'alice' with GRANT OPTION; and she has in turn granted INSERT to user 'carol'.

dbtoyaml currently supports extracting information about nearly all types of Postgres database objects. See *API Reference* for a list of supported objects.

The behavior and options of `dbtoyaml` are patterned after the pg_dump utility since it is most analogous to using `pg_dump --schema-only`.

3.10.4 Multiple File Output

The *--multiple-files* option breaks down the output into multiple files under a given root directory. The root is created if it does not exist. The root directory name defaults to metadata in the system configuration file. The location of the root directory defaults to the configuration item repository.path or can be specified using the *–repository* option (see *Configuration* and *Common Command Line Options* for further details).

The first level contains schema.<name> subdirectories, schema.<name>.yaml files and <objtype>.<name>.yaml files, where <name> is the name of the corresponding objects and <objtype> is the type of top-level (non-schema) object. Note that non-schema refers to Postgres extensions, casts, languages or foreign data wrappers.

The second level, i.e., the schema.<name> subdirectories contain <objtype>.<name>.yaml files for each object in the particular schema (but see below for caveats).

Object Name Conflicts

The names of Postgres objects can include characters that are not allowed in filesystem object names. The most common example is the division operator ('/'), but even table names can include non-alphanumeric characters, if the identifiers are quoted.

In addition, one can define two or more objects with the same base name, e.g., function foo(integer) and function foo(text), or a table named "My Table" and another named "my table" or "MY TABLE". On certain operating systems, i.e., Windows, it is not possible to create two files in the same directory that differ only in the case of their characters.

In order to deal with the aforementioned issues, dbtoyaml places certain objects in common files and transforms object identifiers so that they are suitable for use in files and directories. For example, the information for all user-defined casts are written to the file cast.yaml in the root directory. Functions with the same name but different arguments are written to a single file, e.g., function.foo.yaml in the first example above. Identifiers are also converted to all lowercase, non-alphanumeric characters (excluding underscore) are converted to underscores and, by default, schema object names are truncated to 32 characters.

If two object names, thus transformed, map to the same string, then the objects' information is written to the same file, e.g., table.my_table.yaml in the second example above. If you prefer to change the default truncation length, please define the environment variable PYRSEAS_MAX_IDENT_LEN to some integer value (up to 63).

Version Control and Dropped Objects

It is expected that the output of dbtoyaml --multiple-files will be placed under version control. Further invocations should then update the files in the same directory tree. However, if an object is dropped from the database dbtoyaml would normally only output files for new or changed objects–and thus keep the dropped object file under version control. To deal with dropped objects, dbtoyaml -m outputs a special YAML "index" file, named database.<dbname>.yaml in the root directory. When dbtoyaml -m is run a second time, it looks for this "index" file and if found, proceeds to delete the previous run's .yaml files before outputting new ones.

3.10.5 Options

dbtoyaml accepts the following command-line arguments (in addition to the *Common Command Line Options*):

dbname

> Specifies the name of the database whose schema is to be extracted.

-m, --multiple-files
Extracts the schema to a two-level directory tree. See *Multiple File Output* above.

-n `<schema>`
--schema `<schema>`
> Extracts only a schema matching *schema*. By default, all schemas are extracted. Multiple schemas can be extracted by using multiple −n switches. Note that normally all objects that belong to the schema are extracted as well, unless excluded otherwise.

-N `<schema>`
--exclude-schema `<schema>`
> Does not extract schema matching *schema*. This can be given more than once to exclude several schemas.

-O, --no-owner
> Do not output object ownership information. By default, as seen in the sample output above, database objects (schemas, tables, etc.) that can be owned by some user, are shown with an "owner: *username*" element. The *−O* switch suppresses all those lines.

> NOTE: If you specify *–no-owner*, you will most likely also want to specify *--no-privileges*. If the former is used without the latter the resulting YAML output will have privilege information without user data, which will cause errors if the YAML is then fed to *yamltodb - YAML to Database*.

-t `<table>`
--table `<table>`
> Extract only tables matching *table*. Multiple tables can be extracted by using multiple *−t* switches. Note that selecting a table may cause other objects, such as an owned sequence, to be extracted as well

-T `<table>`
--exclude-table `<table>`
> Do not extract tables matching *table*. Multiple tables can be excluded by using multiple *−T* switches.

-x, --no-privileges
> Do not output access privilege information. By default, as seen in the sample output above, if specific GRANTs have been issued on various objects (schemas, tables, etc.), the privileges are shown under each object. The *−x* switch suppresses all those lines.

> See also the NOTE under *--no-owner*.

3.10.6 Examples

To extract a database called `moviesdb` into a file:

```
dbtoyaml moviesdb > moviesdb.yaml
```

To extract only the schema named `store`:

```
dbtoyaml --schema=store moviesdb > moviesdb.yaml
```

To extract the tables named `film` and `genre`:

```
dbtoyaml -t film -t genre moviesdb -o moviesdb.yaml
```

To extract objects, to standard output, except those in schemas `product` and `store`:

```
dbtoyaml -N product -N store moviesdb
```

To extract objects to a directory under version control:

```
dbtoyaml moviesdb -m movies/dbspec
```

3.11 yamltodb - YAML to Database

3.11.1 Name

yamltodb – generate SQL statements to update a Postgres database to match the schema specified in a YAML file

3.11.2 Synopsys

```
yamltodb [option...] dbname [spec]
```

3.11.3 Description

yamltodb is a utility for generating SQL statements to update a Postgres database so that it will match the schema specified in an input YAML formatted specification file.

For example, given the input file shown under *dbtoyaml - Database to YAML*, **yamltodb**, when run against a newly-created database, outputs the following SQL statements:

```
CREATE SCHEMA s1;
ALTER SCHEMA s1 OWNER TO bob;
GRANT ALL ON SCHEMA s1 TO bob;
GRANT ALL ON SCHEMA s1 TO alice;
CREATE TABLE t1 (
    c1 integer NOT NULL,
    c2 smallint,
    c3 boolean DEFAULT false,
    c4 text);
ALTER TABLE t1 OWNER TO alice;
CREATE TABLE s1.t2 (
    c21 integer NOT NULL,
    c22 character varying(16));
ALTER TABLE s1.t2 OWNER TO bob;
GRANT ALL ON TABLE s1.t2 TO bob;
GRANT SELECT ON TABLE s1.t2 TO PUBLIC;
GRANT INSERT, DELETE, UPDATE ON TABLE s1.t2 TO alice WITH GRANT OPTION;
GRANT INSERT ON TABLE s1.t2 TO carol;
ALTER TABLE t1 ADD CONSTRAINT t1_c2_check CHECK (c2 > 123);
ALTER TABLE t1 ADD CONSTRAINT t1_pkey PRIMARY KEY (c1);
ALTER TABLE s1.t2 ADD CONSTRAINT t2_pkey PRIMARY KEY (c21);
ALTER TABLE t1 ADD CONSTRAINT t1_c2_fkey FOREIGN KEY (c2) REFERENCES s1.t2 (c21);
```

3.11.4 Options

yamltodb accepts the following command-line arguments (in addition to the *Common Command Line Options*):

dbname

> Specifies the name of the database whose schema is to analyzed.

spec

> Specifies the location of the YAML specification. If this is omitted or specified as a single or double dash, the specification is read from the program's standard input. However, if the `--multiple-files` option is used, that takes precedence.

-m, --multiple-files
> Specifies that input should be taken from YAML specification files present in a two-level (metadata) directory tree. See *Multiple File Output* under *dbtoyaml - Database to YAML* for further details.

-n <schema>
--schema <schema>
> Compare only a schema matching *schema*. By default, all schemas are compared. Multiple schemas can be compared by using multiple *-n* switches.

-1
--single-transaction
> Wrap the generated statements in BEGIN/COMMIT. This ensures that either all the statements complete successfully, or no changes are applied.

-u, --update
> Execute the generated statements against the database mentioned in **dbname**. This implies the *--single-transaction* option.

--revert
> Generate SQL in reversion mode, that is, to undo the changes that would normally be generated. For example, if without this option, the SQL would be a DROP TABLE, the *--revert* option generates a CREATE TABLE with all the columns, constraints and other objects associated with the table being dropped.
>
> This option is experimental and currently has only been implemented for schemas and sequences.

3.11.5 Examples

Given a YAML file named moviesdb.yaml, to generate SQL statements to update a database called *mymovies*:

```
yamltodb mymovies moviesdb.yaml
```

To generate the statements as above and immediately update *mymovies*:

```
yamltodb mymovies moviesdb.yaml | psql mymovies
```

or:

```
yamltodb --update mymovies moviesdb.yaml
```

To generate the statements directly from the ouput of **dbtoyaml** (against a different database), with statements enclosed in a single transaction, and save the statements in a file named mymovies.sql:

```
dbtoyaml devmovies | yamltodb -1 mymovies -o mymovies.sql
```

3.12 Common Command Line Options

The Pyrseas utilities support the following command line options:

-c <config-file>
--config <config-file>
> Specifies an additional *configuration file* to be read and merged with configuration information from other sources. See *Configuration* for more details.

-H <host>

--host `<host>`
> Specifies the *host name* of the machine on which the Postgres server is running. The default host name is determined by Postgres (normally, a Unix-domain socket or `localhost`).

-h, --help
> Show help about the program's command line arguments, and exit.

-o `<file>`
--output `<file>`
> Send output to the specified *file*. If this is omitted, the standard output is used.

-p `<port>`
--port `<port>`
> Specifies the *port* on which the Postgres server is listening for connections. The default port number is determined by Postgres (normally, 5432).

-r `<path>`
--repository `<path>`
> Specifies the *path* to a directory where metadata and static data files will be written to or read from, or where an additional configuration file can be found. Normally, this will be the root of a version control repository. If this is not specified on the command line or in a configuration file, it defaults to the current working directory.

-U `<username>`
--user `<username>`
> Postgres *user name* to connect as. The default user name is determined by Postgres (normally, the name of the operating system user running the program).

--version
> Print the program name and version identifier and exit.

-W, --password
> Force the program to prompt for a password before connecting to a database. If this option is not specified and password authentication is required, the program will resort to libpq defaults, i.e., password file or PGPASS-WORD environment variable.

Short options (those only one character long) can be concatenated with their value arguments, e.g.:

```
dbtoyaml -p5433 dbname
```

Several short options can be joined together, using only a single - prefix, as long as only the last option (or none of them) requires a value.

Long options (those with names longer than a single-character) can be separated from their arguments by a '=' or passed as two separate arguments. For example:

```
dbtoyaml --port=5433 dbname
```

or:

```
dbtoyaml --port 5433 dbname
```

Long options can be abbreviated as long as the abbreviation is unambiguous:

```
dbtoyaml --pass dbname
```

API Reference

Currently, the only external APIs are the class *Database* and the methods *to_map()* and *diff_map()* of the latter. Other classes and methods are documented mainly for developer use.

4.1 Database Objects

The dbobject module defines two low-level classes and an intermediate class. Most Pyrseas classes are derived from either *DbObject* or *DbObjectDict*.

4.1.1 Database Object

A *DbObject* represents a database object such as a schema, table, or column, defined in a Postgres system catalog. It is initialized from a dictionary of attributes. Derived classes should define a keylist that is a list of attribute names that uniquely identify each object instance within the database.

class pyrseas.dbobject.**DbObject**(*name*, *description=None*, ***attrs*)
> A single object in a database catalog, e.g., a schema, a table, a column

DbObject.**objtype**
> Type of object as an uppercase string, for SQL syntax generation

> This is used in most CREATE, ALTER and DROP statements. It is also used by *extern_key()* in lowercase form.

DbObject.**keylist = ['name']**
> List of attributes that uniquely identify the object in the catalogs

> See description of *key()* for further details.

DbObject.**key**()
> Return a tuple that identifies the database object

> > **Returns** a single string or a tuple of strings

This is used as key for all internal maps. The first-level objects (schemas, languages and casts) use the object name as the key. Second-level (schema-owned) objects usually use the schema name and the object name as the key. Some object types need longer keys, e.g., operators need schema name, operator symbols, left argument and right argument.

Each class implementing an object type specifies a *keylist* attribute, i.e., a list giving the names of attributes making up the key.

The following methods are generally used to map objects for external output:

DbObject.**extern_key**()
> Return the key to be used in external maps for this object

> > **Returns** string

> This is used for the first two levels of external maps. The first level is the one that includes schemas, as well as extensions, languages, casts and foreign data wrappers. The second level includes all schema-owned objects, i.e., tables, functions, operators, etc. All subsequent levels, e.g., primary keys, indexes, etc., currently use the object name as the external identifier, appearing in the map after an object grouping header, such as primary_key.

> The common format for an external key is *object-type non-schema-qualified-name*, where *object-type* is the lowercase version of *objtype*, e.g., table tablename. Some object types require more, e.g., functions need the signature, so they override this implementation.

static DbObject.**query**(*dbversion=None*)
> The SQL SELECT query to fetch object instances from the catalogs

> > **Parameters dbversion** – Postgres version identifier

> This is used by the method fetch(). The *dbversion* parameter is used in descendant classes to customize the queries according to the target Postgres version.

DbObject.**extern_filename**(*ext='yaml'*, *truncate=False*)
> Return a filename to be used to output external files

> > **Parameters**

> > > • **ext** – file extension

> > > • **truncate** – truncate filename to MAX_IDENT_LEN

> > **Returns** filename string

> This is used for the first two levels of external (metadata) files. The first level is the one that includes schemas, as well as extensions, languages, casts and FDWs. The second level includes all schema-owned objects, i.e., tables, functions, operators, etc.

> The common format for the filename is *objtype.objname.yaml*, e.g., for a table *t1* the filename is "table.t1.yaml". For an object name that has characters not allowed in filesystems, the characters are replaced by underscores.

DbObject.**identifier**()
> Returns a full identifier for the database object

> > **Returns** string

> This is used by *comment()*, *alter_owner()* and *drop()* to generate SQL syntax referring to the object. It does not include the object type, but it may include (in overriden methods) other elements, e.g., the arguments to a function.

DbObject.**to_map**(*db*, *no_owner=False*, *no_privs=False*, *deepcopy=True*)
> Convert an object to a YAML-suitable format

> > **Parameters**

> > > • **db** – db used to tie the objects together

- **no_owner** – exclude object owner information

- **no_privs** – exclude privilege information

>> **Returns** dictionary

> The return value, a Python dictionary, is equivalent to a YAML or JSON object.

DbObject.**map_privs**()
> Return a list of access privileges on the current object

>> **Returns** list

The following methods generate SQL statements from the object properties and sometimes from a second object:

DbObject.**comment**()
> Return SQL statement to create a COMMENT on the object

>> **Returns** SQL statement

DbObject.**alter_owner**(*owner=None*)
> Return ALTER statement to set the OWNER of an object

>> **Returns** SQL statement

DbObject.**drop**()
> Generate SQL to drop the current object

>> **Returns** list of SQL statements

DbObject.**rename**(*oldname*)
> Return SQL statement to RENAME the object

>> **Parameters** **oldname** – the old name for the object

>> **Returns** SQL statement

DbObject.**alter**(*inobj*, *no_owner=False*)
> Generate SQL to transform an existing database object

>> **Parameters** **inobj** – a YAML map defining the new object

>> **Returns** list of SQL statements

> Compares the current object to an input object and generates SQL statements to transform it into the one represented by the input.

DbObject.**diff_privileges**(*inobj*)
> Generate SQL statements to grant or revoke privileges

>> **Parameters** **inobj** – a YAML map defining the input object

>> **Returns** list of SQL statements

DbObject.**diff_description**(*inobj*)
> Generate SQL statements to add or change COMMENTs

>> **Parameters** **inobj** – a YAML map defining the input object

>> **Returns** list of SQL statements

4.1.2 Database Object Dictionary

A *DbObjectDict* represents a collection of *DbObject*'s and is derived from the Python built-in type `dict`. If a DbConnection object is used for initialization, an internal method is called to initialize the dictionary from the database catalogs. The *DbObjectDict* fetch() method fetches all objects using the *query* method defined

---

by derived classes. Derived classes should also define a `cls` attribute for the associated *DbObject* class, e.g., `SchemaDict` sets `cls` to `Schema`.

class `pyrseas.dbobject.`**DbObjectDict**(*dbconn=None*)
> A dictionary of database objects, all of the same type.
>
> However, note that "type" sometimes refers to a polymorphic class. For example, a `ConstraintDict` holds objects of type `Constraint`, but the actual objects may be of class `CheckConstraint`, `PrimaryKey`, etc.

`DbObjectDict.`**cls = <class 'pyrseas.dbobject.DbObject'>**
> The possibly-polymorphic class, derived from *DbObject* that the objects belong to.

`DbObjectDict.`**to_map**(*db*, *opts*)
> Convert the object dictionary to a regular dictionary
>
> > **Parameters**
> >
> > > • **db** – db used to tie the objects together
> > >
> > > • **opts** – options to include/exclude information, etc.
> >
> > **Returns** dictionary
>
> Invokes the *to_map* method of each object to construct the dictionary. If *opts* specifies a directory, the objects are written to files in that directory.

`DbObjectDict.`**fetch**()
> Fetch all objects from the catalogs using the associated `query()` methods.
>
> > **Returns** list of self.cls (polymorphic) objects

4.1.3 Schema Object

A *DbSchemaObject* is derived from *DbObject*. It is used as a base class for objects owned by a schema and to define certain common methods. This is different from the `Schema` that represents the schema (Postgres namespace) itself.

class `pyrseas.dbobject.`**DbSchemaObject**(*name*, *schema='public'*, *description=None*, ***attrs*)
> A database object that is owned by a certain schema

`DbSchemaObject.`**identifier**()
> Return a full identifier for a schema object
>
> > **Returns** string

`DbSchemaObject.`**qualname**(*schema=None*, *objname=None*)
> Return the schema-qualified name of self or a related object
>
> > **Returns** string

`DbSchemaObject.`**unqualify**(*objname*)
> Adjust the object name if it is qualified
>
> > **Parameters** **objname** – object name
> >
> > **Returns** unqualified object name

`DbSchemaObject.`**rename**(*oldname*)
> Return SQL statement to RENAME the schema object
>
> > **Parameters** **oldname** – the old name for the schema object
> >
> > **Returns** SQL statement

4.2 Databases

The `database` module defines class *Database*.

4.2.1 Database

A *Database* can be viewed as a tree of database objects. The tree may have one or two main branches. A tree with one main branch is used by **dbtoyaml** to hold the representation of the database, as read from the Postgres catalogs. **yamltodb** uses a second main branch to hold the representation as read from the YAML input specification.

Each main branch consists of multiple subtrees for different kinds of objects. For example, the Schemas (Postgres namespaces) subtree has all the Postgres schema objects, the Procedures subtree has all the Postgres functions and aggregates. The objects in the subtrees are connected in implicit or explicit manners to related objects. For example, the objects in the `schema public` are implicitly accessible from the corresponding `Schema` object because they all share `public` as the first part of their internal key (see `DbObject.key()`). As another example, a table has explicit links to constraints and indexes defined on it.

A *Database* is initialized from a `CatDbConnection` object (a specialized class derived from `pgdbconn.dbconn.DbConnection`). It consists of one or two `Dicts` (the main branches in the above discussion). A `Dicts` object holds various dictionary objects derived from *DbObjectDict*, e.g., *SchemaDict*, *ClassDict*, and *ColumnDict*. The key for each dictionary is a Python tuple (or a single value in the case of `SchemaDict` and other non-schema objects). For example, the *ClassDict* dictionary is indexed by (*schema name*, *table name*)–in this context *table name* may actually be a *sequence name*, a *view name* or a *materialized view name*. In addition, object instances in each dictionary are linked to related objects in other dictionaries, e.g., columns are linked to the tables where they belong.

The `db` `Dicts` object –always present– instantiates the database schemas, including their tables and other objects, by querying the system catalogs. The `ndb` `Dicts` object instantiates the schemas based on the `input_map` supplied to the `diff_map()` method.

The `to_map()` method returns and the `diff_map()` method takes as input, a Python dictionary (equivalent to a YAML or JSON object) as shown below. It uses 'schema *schema_name*' as the key for each schema. The value corresponding to each 'schema *schema_name*' is another dictionary using 'sequences', 'tables', etc., as keys and more dictionaries as values. For example:

```
{'schema public':
    {'sequence seq1': { ... },
     'sequence seq2': { ... },
     'table t1': { ... },
     'table t2': { ... },
     'table t3': { ... },
     'view v1': { ... }
    },
 'schema s1': { ... },
 'schema s2': { ... }
}
```

Refer to *Sequence*, *Table* and `View` for details on the lower level dictionaries.

class pyrseas.database.**Database**(*config*)
    A database definition, from its catalogs and/or a YAML spec.

Methods `from_catalog()` and `from_map()` are for internal use. Methods `to_map()` and `diff_map()` are the external API.

Database.**from_catalog**(*single_db=False*)
    Populate the database objects by querying the catalogs

> > **Parameters single_db** – populating only this database?

> The *db* holder is populated by various DbObjectDict-derived classes by querying the catalogs. A dependency graph is constructed by querying the pg_depend catalog. The objects in the dictionary are then linked to related objects, e.g., columns are linked to the tables they belong.

`Database.from_map`(*input_map*, *langs=None*)
> Populate the new database objects from the input map

> > **Parameters**

> > > • **input_map** – a YAML map defining the new database

> > > • **langs** – list of language templates

> The *ndb* holder is populated by various DbObjectDict-derived classes by traversing the YAML input map. The objects in the dictionary are then linked to related objects, e.g., columns are linked to the tables they belong.

`Database.map_from_dir`()
> Read the database maps starting from the metadata directory

> > **Returns** dictionary

`Database.to_map`()
> Convert the db maps to a single hierarchy suitable for YAML

> > **Returns** a YAML-suitable dictionary (without any Python objects)

`Database.diff_map`(*input_map*, *quote_reserved=True*)
> Generate SQL to transform an existing database

> > **Parameters**

> > > • **input_map** – a YAML map defining the new database

> > > • **quote_reserved** – fetch reserved words

> > **Returns** list of SQL statements

> Compares the existing database definition, as fetched from the catalogs, to the input YAML map and generates SQL statements to transform the database into the one represented by the input.

4.3 Schemas

The `schema` module defines two classes, *Schema* and *SchemaDict*, derived from `DbObject` and `DbObjectDict`, respectively.

4.3.1 Schema

Schema is derived from *DbObject* and represents a database schema or Postgres namespace, i.e., a collection of tables and other objects. The 'public' schema is currently treated specially as in most contexts an unqualified object is assumed to be part of it, e.g., table "t" is usually shorthand for table "public.t." The 'pyrseas' schema, if present, is excluded as it is only intended for use by **dbaugment** or other Pyrseas internal purposes.

class `pyrseas.dbobject.schema.Schema`(*name*, *description=None*, *owner=None*, *privileges=[]*,
> > *oldname=None*, *oid=None*)
> A database schema definition, i.e., a named collection of tables, views, triggers and other schema objects.

`Schema.extern_dir`(*root='.'*)
> Return the path to a directory to hold the schema objects.

> **Returns** directory path

Schema.**to_map**(*db*, *dbschemas*, *opts*)
> Convert tables, etc., dictionaries to a YAML-suitable format

> > **Parameters**
> >
> > > - **dbschemas** – dictionary of schemas
> > >
> > > - **opts** – options to include/exclude schemas/tables, etc.
> >
> > **Returns** dictionary

Schema.**create**(**args*, ***kwargs*)
> Return SQL statements to CREATE the schema

> > **Returns** SQL statements

Schema.**drop**()
> Generate SQL to drop the current object

> > **Returns** list of SQL statements

Schema.**data_import**(*opts*)
> Generate SQL to import data from the tables in this schema

> > **Parameters** **opts** – options to include/exclude schemas/tables, etc.

> > **Returns** list of SQL statements

4.3.2 Schema Dictionary

SchemaDict is derived from *DbObjectDict*. It is a dictionary that represents the collection of schemas in a database. Certain internal schemas (information_schema, pg_catalog, etc.) owned by the 'postgres' user are excluded.

class pyrseas.dbobject.schema.**SchemaDict**(*dbconn=None*)
> The collection of schemas in a database. Minimally, the 'public' schema.

Method from_map() is called from Database *from_map()* to start a recursive interpretation of the input map. The inmap argument is the same as input to the *diff_map()* method of Database. The newdb argument is the holder of *DbObjectDict*-derived dictionaries which is filled in as the recursive interpretation proceeds.

SchemaDict.**from_map**(*inmap*, *newdb*)
> Initialize the dictionary of schemas by converting the input map

> > **Parameters**
> >
> > > - **inmap** – the input YAML map defining the schemas
> > >
> > > - **newdb** – collection of dictionaries defining the database

> Starts the recursive analysis of the input map and construction of the internal collection of dictionaries describing the database objects.

SchemaDict.**to_map**(*db*, *opts*)
> Convert the schema dictionary to a regular dictionary

> > **Parameters** **opts** – options to include/exclude schemas/tables, etc.

> > **Returns** dictionary

> Invokes the *to_map* method of each schema to construct a dictionary of schemas.

SchemaDict.**data_import**(*opts*)
>   Iterate over schemas with tables to be imported

>   > **Parameters** **opts** – options to include/exclude schemas/tables, etc.

>   > **Returns** list of SQL statements

4.4 Non-schema Objects

4.4.1 Casts

The `cast` module defines two classes, *Cast* and *CastDict*, derived from DbObject and DbObjectDict, respectively.

Cast

Cast is derived from *DbObject* and represents a [Postgres cast](#).

A cast is identified externally as `cast (<source_type> AS <target_type>)`.

class pyrseas.dbobject.cast.**Cast**(*source*, *target*, *description*, *function*, *context*, *method*, *oid=None*)
>   A cast from a source type to a target type

Cast.**extern_key**()
>   Return the key to be used in external maps for this cast

>   > **Returns** string

Cast.**identifier**()
>   Return a full identifier for a cast object

>   > **Returns** string

Cast.**to_map**(*db*, *no_owner=False*, *no_privs=False*)
>   Convert a cast to a YAML-suitable format

>   > **Returns** dictionary

Cast.**create**(**args*, ***kwargs*)
>   Return SQL statements to CREATE the cast

>   > **Returns** SQL statements

Cast Dictionary

CastDict is derived from *DbObjectDict*. It is a dictionary that represents the collection of casts in a database.

class pyrseas.dbobject.cast.**CastDict**(*dbconn=None*)
>   The collection of casts in a database

CastDict.**from_map**(*incasts*, *newdb*)
>   Initalize the dictionary of casts by converting the input map

>   > **Parameters**

>   > - **incasts** – YAML map defining the casts

>   > - **newdb** – collection of dictionaries defining the database

---

4.4.2 Event Triggers

The `eventtrig` module defines two classes, *EventTrigger* and *EventTriggerDict*, derived from `DbObject` and `DbObjectDict`, respectively.

Event Trigger

EventTrigger is derived from *DbObject* and represents an event trigger available from Postgres 9.3 onwards.

class pyrseas.dbobject.eventtrig.**EventTrigger**(*name*, *description*, *owner*, *event*, *procedure*, *enabled=False*, *tags=None*, *oid=None*)

> An event trigger

EventTrigger.**to_map**(*db*, *no_owner=False*, *no_privs=False*)
> Convert an event trigger definition to a YAML-suitable format

> > **Parameters** **db** – db used to tie the objects together

> > **Returns** dictionary

EventTrigger.**create**(**args*, ***kwargs*)
> Return SQL statements to CREATE the event trigger

> > **Returns** SQL statements

Event Trigger Dictionary

EventTriggerDict is derived from *DbObjectDict*. It is a dictionary that represents the collection of event triggers in a database.

class pyrseas.dbobject.eventtrig.**EventTriggerDict**(*dbconn=None*)
> The collection of event triggers in a database

EventTriggerDict.**from_map**(*intriggers*, *newdb*)
> Initalize the dictionary of triggers by converting the input map

> > **Parameters**

> > > • **intriggers** – YAML map defining the event triggers

> > > • **newdb** – dictionary of input database

4.4.3 Extensions

The `extension` module defines two classes, *Extension* and *ExtensionDict*, derived from `DbObject` and `DbObjectDict`, respectively.

Extension

Extension is derived from *DbObject* and represents a Postgres extension.

class pyrseas.dbobject.extension.**Extension**(*name*, *description*, *owner*, *schema*, *version=None*, *oid=None*)

> An extension

Extension.**create**(**args*, ***kwargs*)
> Return SQL statements to CREATE the extension

> **Returns** SQL statements

Extension.**alter**(*inobj*, *no_owner=True*)

> Generate SQL to transform an existing extension

> > **Parameters inobj** – a YAML map defining the new extension

> > **Returns** list of SQL statements

> This exists because ALTER EXTENSION does not permit altering the owner.

Extension Dictionary

ExtensionDict is derived from *DbObjectDict*. It is a dictionary that represents the collection of extensions in a database.

class pyrseas.dbobject.extension.**ExtensionDict**(*dbconn=None*)

> The collection of extensions in a database

ExtensionDict.**from_map**(*inexts*, *langtempls*, *newdb*)

> Initalize the dictionary of extensions by converting the input map

> > **Parameters**

> > - **inexts** – YAML map defining the extensions
> > - **langtempls** – list of language templates
> > - **newdb** – dictionary of input database

4.4.4 Foreign Data Objects

The `foreign` module defines nine classes related to Postgres foreign data wrappers (FDWs), namely: *DbObjectWithOptions* derived from DbObject, classes *ForeignDataWrapper*, *ForeignServer* and *UserMapping* derived from *DbObjectWithOptions*, *ForeignTable* derived from *DbObjectWithOptions* and Table, classes *ForeignDataWrapperDict*, *ForeignServerDict* and *UserMappingDict* derived from DbObjectDict, and *ForeignTableDict* derived from ClassDict.

Database Object With Options

DbObjectWithOptions is derived from *DbObject*. It is a helper class for dealing with the OPTIONS clauses common to the foreign data objects.

class pyrseas.dbobject.foreign.**DbObjectWithOptions**(*name*, *options*)

> Helper class for database objects with OPTIONS clauses

DbObjectWithOptions.**to_map**(*db*, *no_owner=False*, *no_privs=False*)

> Convert objects to a YAML-suitable format

> > **Parameters**

> > - **no_owner** – exclude object owner information
> > - **no_privs** – exclude privilege information

> > **Returns** dictionary

DbObjectWithOptions.**options_clause**()

> Create the OPTIONS clause

> > **Parameters optdict** – the dictionary of options

Returns SQL OPTIONS clause

DbObjectWithOptions.**diff_options**(*newopts*)
    Compare options lists and generate SQL OPTIONS clause

        Newopts list of new options

        Returns SQL OPTIONS clause

    Generate ([ADD|SET|DROP key 'value') clauses from two lists in the form of 'key=value' strings.

DbObjectWithOptions.**alter**(*inobj*)
    Generate SQL to transform an existing object with options

        Parameters inobj – a YAML map defining the new object

        Returns list of SQL statements

Foreign Data Wrapper

ForeignDataWrapper is derived from *DbObjectWithOptions* and represents a Postgres foreign data wrapper. See also Foreign Data and Writing A Foreign Data Wrapper.

class pyrseas.dbobject.foreign.**ForeignDataWrapper**(*name*, *options*, *description*, *owner*, *privileges*, *handler=None*, *validator=None*, *oid=None*)

    A foreign data wrapper definition

ForeignDataWrapper.**to_map**(*db*, *no_owner*, *no_privs*)
    Convert wrappers and subsidiary objects to a YAML-suitable format

        Parameters

            • **no_owner** – exclude object owner information

            • **no_privs** – exclude privilege information

        Returns dictionary

ForeignDataWrapper.**create**(**args*, ***kwargs*)
    Return SQL statements to CREATE the data wrapper

        Returns SQL statements

Foreign Data Wrapper Dictionary

ForeignDataWrapperDict is derived from *DbObjectDict*. It is a dictionary that represents the collection of foreign data wrappers in a database.

class pyrseas.dbobject.foreign.**ForeignDataWrapperDict**(*dbconn=None*)
    The collection of foreign data wrappers in a database

ForeignDataWrapperDict.**from_map**(*inwrappers*, *newdb*)
    Initialize the dictionary of wrappers by examining the input map

        Parameters

            • **inwrappers** – input YAML map defining the data wrappers

            • **newdb** – collection of dictionaries defining the database

---

Foreign Server

ForeignServer is derived from *DbObjectWithOptions* and represents a Postgres foreign server.

class pyrseas.dbobject.foreign.**ForeignServer**(*name*, *options*, *description*, *owner*, *privileges*, *wrapper*, *type=None*, *version=None*, *oid=None*)

> A foreign server definition

ForeignServer.**identifier**()
> Returns a full identifier for the foreign server

> > **Returns** string

ForeignServer.**to_map**(*db*, *no_owner*, *no_privs*)
> Convert servers and subsidiary objects to a YAML-suitable format

> > **Parameters**

> > > • **no_owner** – exclude server owner information

> > > • **no_privs** – exclude privilege information

> > **Returns** dictionary

ForeignServer.**create**(**args*, ***kwargs*)
> Return SQL statements to CREATE the server

> > **Returns** SQL statements

Foreign Server Dictionary

ForeignServerDict is derived from *DbObjectDict*. It is a Python dictionary that represents the collection of foreign servers in a database.

class pyrseas.dbobject.foreign.**ForeignServerDict**(*dbconn=None*)
> The collection of foreign servers in a database

ForeignServerDict.**from_map**(*wrapper*, *inservers*, *newdb*)
> Initialize the dictionary of servers by examining the input map

> > **Parameters**

> > > • **wrapper** – associated foreign data wrapper

> > > • **inservers** – input YAML map defining the foreign servers

> > > • **newdb** – collection of dictionaries defining the database

ForeignServerDict.**to_map**(*db*, *no_owner*, *no_privs*)
> Convert the server dictionary to a regular dictionary

> > **Parameters**

> > > • **no_owner** – exclude server owner information

> > > • **no_privs** – exclude privilege information

> > **Returns** dictionary

> Invokes the *to_map* method of each server to construct a dictionary of foreign servers.

User Mapping

UserMapping is derived from *DbObjectWithOptions* and represents a mapping of a Postgres user to a foreign server.

class pyrseas.dbobject.foreign.**UserMapping**(*name*, *options*, *wrapper*, *server*, *oid=None*)
A user mapping definition

UserMapping.**extern_key**()
Return the key to be used in external maps for this user mapping

> **Returns** string

UserMapping.**identifier**()
Return a full identifier for a user mapping object

> **Returns** string

UserMapping.**create**(*dbversion=None*)
Return SQL statements to CREATE the user mapping

> **Returns** SQL statements

User Mapping Dictionary

UserMappingDict is derived from *DbObjectDict*. It is a dictionary that represents the collection of user mappings in a database.

class pyrseas.dbobject.foreign.**UserMappingDict**(*dbconn=None*)
The collection of user mappings in a database

UserMappingDict.**from_map**(*server*, *inusermaps*)
Initialize the dictionary of mappings by examining the input map

> **Parameters**
>
> - **server** – foreign server associated with mappings
>
> - **inusermaps** – input YAML map defining the user mappings

UserMappingDict.**to_map**(*db*)
Convert the user mapping dictionary to a regular dictionary

> **Returns** dictionary

Invokes the *to_map* method of each mapping to construct a dictionary of user mappings.

Foreign Table

ForeignTable is derived from *DbObjectWithOptions* and *Table*. It represents a Postgres foreign table.

class pyrseas.dbobject.foreign.**ForeignTable**(*name*, *schema*, *description*, *owner*, *privileges*, *server=None*, *options={}*, *oid=None*)
A foreign table definition

ForeignTable.**to_map**(*db*, *opts*)
Convert a foreign table to a YAML-suitable format

> **Parameters** **opts** – options to include/exclude tables, etc.

> **Returns** dictionary

ForeignTable.**create**(**args*, ***kwargs*)
> Return SQL statements to CREATE the foreign table
>
>> **Returns** SQL statements

ForeignTable.**drop**()
> Return a SQL DROP statement for the foreign table
>
>> **Returns** SQL statement

Foreign Table Dictionary

ForeignTableDict is derived from *ClassDict*. It is a dictionary that represents the collection of foreign tables in a database.

class pyrseas.dbobject.foreign.**ForeignTableDict**(*dbconn=None*)
> The collection of foreign tables in a database

ForeignTableDict.**from_map**(*schema*, *inobjs*, *newdb*)
> Initalize the dictionary of tables by converting the input map
>
>> **Parameters**
>>
>> - **schema** – schema owning the tables
>>
>> - **inobjs** – YAML map defining the schema objects
>>
>> - **newdb** – collection of dictionaries defining the database

4.4.5 Procedural Languages

The language module defines two classes, *Language* and *LanguageDict*, derived from DbObject and DbObjectDict, respectively.

Procedural Language

Language is derived from *DbObject* and represents a procedural language.

class pyrseas.dbobject.language.**Language**(*name*, *description=None*, *owner=None*, *privileges=[]*, *trusted=False*, *oid=None*)
> A procedural language definition

Language.**to_map**(*db*, *no_owner*, *no_privs*)
> Convert language to a YAML-suitable format
>
>> **Parameters** **no_owner** – exclude language owner information
>>
>> **Returns** dictionary

Language.**drop**()
> Generate SQL to drop the current object
>
>> **Returns** list of SQL statements

Language.**create**(*dbversion=None*)
> Return SQL statements to CREATE the language
>
>> **Returns** SQL statements

Language Dictionary

LanguageDict is derived from *DbObjectDict*. It is a dictionary that represents the collection of procedural languages in a database. Internal languages ('internal', 'c' and 'sql') are excluded.

class pyrseas.dbobject.language.**LanguageDict**(*dbconn=None*)
> The collection of procedural languages in a database.

LanguageDict.**from_map**(*inmap*)
> Initialize the dictionary of languages by examining the input map

> > **Parameters** `inmap` – the input YAML map defining the languages

4.5 Tables and Related Objects

4.5.1 Tables, Views and Sequences

The `table` and `view` modules define six classes, *DbClass* derived from DbSchemaObject, classes *Sequence*, *Table* and View derived from *DbClass*, MaterializedView derived from View, and *ClassDict*, derived from DbObjectDict.

Database Class

Class *DbClass* is derived from *DbSchemaObject* and represents a table, view or sequence as defined in the Postgres *pg_class* catalog.

class pyrseas.dbobject.table.**DbClass**(*name*, *schema*, *description*, *owner*, *privileges*)
> A table, sequence or view

> The *pg_class* catalog also includes Postgres indexes, but for now, indexes have not been implemented as part of the *DbClass* hierarchy.

Sequence

Class *Sequence* is derived from *DbClass* and represents a sequence generator. Its `keylist` attributes are the schema name and the sequence name.

The map returned by `to_map()` and expected as argument by *ClassDict.from_map()* has the following structure:

```
{'sequence seq1':
    {'cache_value': 1,
     'data_type': 'integer',
     'increment_by': 1,
     'max_value': None,
     'min_value': None,
     'owner_column': 'c1',
     'owner_table': 't1',
     'start_value': 1
    }
}
```

class pyrseas.dbobject.table.**Sequence**(*name, schema, description, owner, privileges, start_value=1, increment_by=1, max_value=9223372036854775807, min_value=1, cache_value=1, data_type='bigint', owner_table=None, owner_column=None, oid=None*)

> A sequence generator definition

Sequence.**get_attrs**(*dbconn*)

> Get the attributes for the sequence
>
> > **Parameters dbconn** – a DbConnection object

Sequence.**get_dependent_table**(*dbconn*)

> Get the table and column name that uses or owns the sequence
>
> > **Parameters dbconn** – a DbConnection object

Sequence.**to_map**(*db, opts*)

> Convert a sequence definition to a YAML-suitable format
>
> > **Parameters opts** – options to include/exclude tables, etc.
> >
> > **Returns** dictionary

Sequence.**create**(**args, **kwargs*)

> Return a SQL statement to CREATE the sequence
>
> > **Returns** SQL statements

Sequence.**add_owner**()

> Return statement to ALTER the sequence to indicate its owner table
>
> > **Returns** SQL statement

Sequence.**alter**(*inseq, no_owner=False*)

> Generate SQL to transform an existing sequence
>
> > **Parameters inseq** – a YAML map defining the new sequence
> >
> > **Returns** list of SQL statements
>
> Compares the sequence to an input sequence and generates SQL statements to transform it into the one represented by the input.

Sequence.**drop**()

> Generate SQL to drop the current sequence
>
> > **Returns** list of SQL statements

Table

Class *Table* is derived from *DbClass* and represents a database table. Its keylist attributes are the schema name and the table name.

The map returned by to_map() and expected as argument by *ClassDict.from_map()* has a structure similar to the following:

```
{'table t1':
    {'columns':
        [
        {'c1': {'type': 'integer', 'not_null': True}},
```

(continues on next page)

```
                {'c2': {'type': 'text'}},
                {'c3': {'type': 'smallint'}},
                {'c4': {'type': 'date', 'default': 'now()'}}
                ],
        'description': "this is the comment for table t1",
        'primary_key':
            {'t1_prim_key':
                {'columns': ['c1', 'c2']}
            },
        'foreign_keys':
            {'t1_fgn_key1':
                {'columns': ['c2', 'c3'],
                    'references':
                        {'table': 't2', 'columns': ['pc2', 'pc1']}
                },
             't1_fgn_key2':
                {'columns': ['c2'],
                    'references': {'table': 't3', 'columns': ['qc1']}
                }
            },
        'unique_constraints': {...},
        'indexes': {...}
    }
}
```

The values for `unique_constraints` and `indexes` follow a pattern similar to `primary_key`, but there can be more than one such specification.

class pyrseas.dbobject.table.**Table**(*name*, *schema*, *description*, *owner*, *privileges*, *tablespace=None*, *unlogged=False*, *options=None*, *partition_bound_spec=None*, *partition_by=None*, *partition_cols=None*, *partition_exprs=None*, *oid=None*)

> A database table definition
>
> A table is identified by its schema name and table name. It should have a list of columns. It may have a primary_key, zero or more foreign_keys, zero or more unique_constraints, and zero or more indexes.
>
> A *Table* can also represent a partitioned table or a partition of a partitioned table. The latter's columns are all inherited from the parent (partitioned) table, so they are not shown in an output map (or expected on input).

Table.**column_names**()

> Return a list of column names in the table
>
> > **Returns** list

Table.**to_map**(*db*, *dbschemas*, *opts*)

> Convert a table to a YAML-suitable format
>
> > **Parameters**
> >
> > - **dbschemas** – database dictionary of schemas
> >
> > - **opts** – options to include/exclude tables, etc.
> >
> > **Returns** dictionary

Table.**create**(*dbversion=None*)

> Return SQL statements to CREATE the table
>
> > **Returns** SQL statements

Table.**drop**()
>   Return a SQL DROP statement for the table

>   >   **Returns** SQL statement

Table.**diff_options**(*newopts*)
>   Compare options lists and generate SQL SET or RESET clause

>   >   **Newopts** list of new options

>   >   **Returns** SQL SET / RESET clauses

>   Generate ([SET|RESET storage_parameter=value) clauses from two lists in the form of 'key=value' strings.

Table.**alter**(*intable*)
>   Generate SQL to transform an existing table

>   >   **Parameters** **intable** – a YAML map defining the new table

>   >   **Returns** list of SQL statements

>   Compares the table to an input table and generates SQL statements to transform it into the one represented by
>   the input.

Table.**alter_drop_columns**(*intable*)
>   Generate SQL to drop columns from an existing table

>   >   **Parameters** **intable** – a YAML map defining the new table

>   >   **Returns** list of SQL statements

>   Compares the table to an input table and generates SQL statements to drop any columns missing from the one
>   represented by the input.

Table.**data_export**(*dbconn*, *dirpath*)
>   Copy table data out to a file

>   >   **Parameters**

>   >   >   • **dbconn** – database connection to use

>   >   >   • **dirpath** – full path to the directory for the file to be created

Table.**data_import**(*dirpath*)
>   Generate SQL to import data into a table

>   >   **Parameters** **dirpath** – full path for the directory for the file

>   >   **Returns** list of SQL statements

Class Dictionary

Class *ClassDict* is derived from *DbObjectDict* and represents the collection of tables, views and sequences in
a database.

class pyrseas.dbobject.table.**ClassDict**(*dbconn=None*)
>   The collection of tables and similar objects in a database

ClassDict.**from_map**(*schema*, *inobjs*, *newdb*)
>   Initalize the dictionary of tables by converting the input map

>   >   **Parameters**

>   >   >   • **schema** – schema owning the tables

>   >   >   • **inobjs** – YAML map defining the schema objects

> • **newdb** – collection of dictionaries defining the database

View

Class *View* is derived from `DbClass` and represents a database view. Its `keylist` attributes are the schema name and the view name.

The map returned by `to_map()` and expected as argument by `ClassDict.from_map()` has a structure similar to the following:

```
{'view v1':
    {'columns': [{'c1': {'type': 'integer'}},
                {'c2': {'type': 'date'}}],
     'definition': " SELECT ...;",
     'description': "this is the comment for view v1"
    }
}
```

class pyrseas.dbobject.view.**View**(*name*, *schema*, *description*, *owner*, *privileges*, *definition*, *oid=None*)

> A database view definition
>
> A view is identified by its schema name and view name.

View.**to_map**(*db*, *opts*)

> Convert a view to a YAML-suitable format
>
> > **Parameters opts** – options to include/exclude tables, etc.
> >
> > **Returns** dictionary

View.**create**(**args*, ***kwargs*)

> Return SQL statements to CREATE the view
>
> > **Returns** SQL statements

View.**alter**(*inview*, *dbversion=None*)

> Generate SQL to transform an existing view
>
> > **Parameters inview** – a YAML map defining the new view
> >
> > **Returns** list of SQL statements
>
> Compares the view to an input view and generates SQL statements to transform it into the one represented by the input.

Materialized View

Class *MaterializedView* is derived from *View* and represents a materialized view. Its `keylist` attributes are the schema name and the view name.

class pyrseas.dbobject.view.**MaterializedView**(*name*, *schema*, *description*, *owner*, *privileges*, *definition*, *with_data=False*, *oid=None*)

> A materialized view definition
>
> A materialized view is identified by its schema name and view name.

MaterializedView.**to_map**(*db*, *opts*)

> Convert a materialized view to a YAML-suitable format

> **Parameters opts** – options to include/exclude tables, etc.
>
> **Returns** dictionary

MaterializedView.**create**(**args*, ***kwargs*)
> Return SQL statements to CREATE the materialized view
>
> > **Returns** SQL statements

4.5.2 Columns

The `column` module defines two classes, *`Column`* derived from `DbSchemaObject` and *`ColumnDict`*, derived from `DbObjectDict`.

Column

`Column` is derived from *`DbSchemaObject`* and represents a column of a table, or an attribute of a composite type. Its `keylist` attributes are the schema name and the table name.

A *`Column`* has the following attributes: `name`, `type`, `not_null`, `default` and `collation`. The `number` attribute is also present but is not made visible externally.

class pyrseas.dbobject.column.**Column**(*name*, *schema*, *table*, *number*, *type*, *description=None*, *privileges=[]*, *not_null=True*, *default=None*, *identity=None*, *collation=None*, *statistics=None*, *inherited=False*, *dropped=False*)
> A table column or attribute of a composite type

Column.**to_map**(*db*, *no_privs*)
> Convert a column to a YAML-suitable format
>
> > **Parameters no_privs** – exclude privilege information
> >
> > **Returns** dictionary

Column.**add**()
> Return a string to specify the column in a CREATE or ALTER TABLE
>
> > **Returns** partial SQL statement

Column.**add_privs**()
> Generate SQL statements to grant privileges on new column
>
> > **Returns** list of SQL statements

Column.**diff_privileges**(*incol*)
> Generate SQL statements to grant or revoke privileges
>
> > **Parameters incol** – a YAML map defining the input column
> >
> > **Returns** list of SQL statements

Column.**comment**()
> Return a SQL COMMENT statement for the column
>
> > **Returns** SQL statement

Column.**drop**()
> Return string to drop the column via ALTER TABLE
>
> > **Returns** SQL statement

Column.**rename**(*newname*)
> Return SQL statement to RENAME the column
>
>> **Parameters** **newname** – the new name of the object
>>
>> **Returns** SQL statement

Column.**alter**(*incol*)
> Generate SQL to transform an existing column
>
>> **Parameters** **insequence** – a YAML map defining the new column
>>
>> **Returns** list of partial SQL statements
>
> Compares the column to an input column and generates partial SQL statements to transform it into the one represented by the input.

Column Dictionary

Class *ColumnDict* is a dictionary derived from *DbObjectDict* and represents the collection of columns in a database, across multiple tables. It is indexed by the schema name and table name, and each value is a list of *Column* objects.

class pyrseas.dbobject.column.**ColumnDict**(*dbconn=None*)
> The collection of columns in tables in a database

ColumnDict.**from_map**(*table*, *incols*)
> Initialize the dictionary of columns by converting the input list
>
>> **Parameters**
>>
>> - **table** – table or type owning the columns/attributes
>> - **incols** – YAML list defining the columns

4.5.3 Constraints

The constraint module defines six classes: *Constraint* derived from DbSchemaObject, classes *CheckConstraint*, *PrimaryKey*, *ForeignKey* and *UniqueConstraint* derived from *Constraint*, and *ConstraintDict* derived from DbObjectDict.

Constraint

Class *Constraint* is derived from *DbSchemaObject* and represents a constraint on a database table. Its keylist attributes are the schema name, the table name and the constraint name.

class pyrseas.dbobject.constraint.**Constraint**(*name*, *schema*, *table*, *description*)
> A constraint definition, such as a primary key, foreign key or unique constraint. This also covers check constraints on domains.

Constraint.**key_columns**()
> Return comma-separated list of key column names
>
>> **Returns** string

Constraint.**add**(**args*, ***kwargs*)
> Return string to add the constraint via ALTER TABLE
>
>> **Returns** SQL statement

Works as is for primary keys and unique constraints but has to be overridden for check constraints and foreign keys.

Constraint.**drop**()
> Return string to drop the constraint via ALTER TABLE

>> **Returns** SQL statement

Constraint.**comment**()
> Return SQL statement to create COMMENT on constraint

>> **Returns** SQL statement

Check Constraint

CheckConstraint is derived from *Constraint* and represents a CHECK constraint.

class pyrseas.dbobject.constraint.**CheckConstraint**(*name*, *schema*, *table*, *description*, *columns*, *expression*, *is_domain_check=False*, *inherited=False*, *oid=None*)

> A check constraint definition

CheckConstraint.**to_map**(*db*, *dbcols*)
> Convert a check constraint definition to a YAML-suitable format

>> **Parameters dbcols** – dictionary of dbobject columns

>> **Returns** dictionary

CheckConstraint.**add**(**args*, ***kwargs*)
> Return string to add the CHECK constraint via ALTER TABLE

>> **Returns** SQL statement

CheckConstraint.**drop**()
> Return string to drop the constraint via ALTER TABLE

>> **Returns** SQL statement

CheckConstraint.**alter**(*inchk*)
> Generate SQL to transform an existing CHECK constraint

>> **Parameters inchk** – a YAML map defining the new CHECK constraint

>> **Returns** list of SQL statements

Compares the CHECK constraint to an input constraint and generates SQL statements to transform it into the one represented by the input.

Primary Key

PrimaryKey is derived from *Constraint* and represents a primary key constraint.

class pyrseas.dbobject.constraint.**PrimaryKey**(*name*, *schema*, *table*, *description*, *columns*, *access_method='btree'*, *tablespace=None*, *cluster=False*, *inherited=False*, *deferrable=False*, *deferred=False*, *oid=None*)

> A primary key constraint definition

`PrimaryKey.`**`to_map`**`(`*db*, *dbcols*`)`

> Convert a primary key definition to a YAML-suitable format
>
> > **Parameters** **`dbcols`** – dictionary of dbobject columns
> >
> > **Returns** dictionary

`PrimaryKey.`**`alter`**`(`*inpk*`)`

> Generate SQL to transform an existing primary key
>
> > **Parameters** **`inpk`** – a YAML map defining the new primary key
> >
> > **Returns** list of SQL statements
>
> Compares the primary key to an input primary key and generates SQL statements to transform it into the one represented by the input.

Foreign Key

ForeignKey is derived from *Constraint* and represents a foreign key constraint.

The following shows a foreign key segment of a map returned by `to_map()` and expected as argument by *ConstraintDict.from_map()* exemplifying various possibilities:

```
{'t1_fgn_key1':
    {
    'columns': ['c2', 'c3'],
    'on_delete': 'restrict',
    'on_update': 'set null',
    'references':
        {'columns': ['pc2', 'pc1'], 'schema': 's1', 'table': 't2'}
    }
}
```

class `pyrseas.dbobject.constraint.`**`ForeignKey`**`(`*name*, *schema*, *table*, *description*, *columns*, *ref_table*, *ref_cols*, *on_update*, *on_delete*, *match*, *access_method='btree'*, *tablespace=None*, *cluster=False*, *inherited=False*, *deferrable=False*, *deferred=False*, *oid=None*`)`

> A foreign key constraint definition

`ForeignKey.`**`ref_columns`**`()`

> Return comma-separated list of reference column names
>
> > **Returns** string

`ForeignKey.`**`to_map`**`(`*db*, *dbcols*, *refcols*`)`

> Convert a foreign key definition to a YAML-suitable format
>
> > **Parameters** **`dbcols`** – dictionary of dbobject columns
> >
> > **Returns** dictionary

`ForeignKey.`**`add`**`(`**args*, ***kwargs*`)`

> Return string to add the foreign key via ALTER TABLE
>
> > **Returns** SQL statement

`ForeignKey.`**`alter`**`(`*infk*`)`

> Generate SQL to transform an existing foreign key

> **Parameters infk** – a YAML map defining the new foreign key
>
> **Returns** list of SQL statements

Compares the foreign key to an input foreign key and generates SQL statements to transform it into the one represented by the input.

Unique Constraint

UniqueConstraint is derived from *Constraint* and represents a UNIQUE, non-primary key constraint.

class pyrseas.dbobject.constraint.**UniqueConstraint**(*name*, *schema*, *table*, *description*, *columns*, *access_method='btree'*, *tablespace=None*, *cluster=False*, *inherited=False*, *deferrable=False*, *deferred=False*, *oid=None*)

> A unique constraint definition

UniqueConstraint.**to_map**(*db*, *dbcols*)
> Convert a unique constraint definition to a YAML-suitable format
>
> > **Parameters dbcols** – dictionary of dbobject columns
> >
> > **Returns** dictionary

UniqueConstraint.**alter**(*inuc*)
> Generate SQL to transform an existing unique constraint
>
> > **Parameters inuc** – a YAML map defining the new unique constraint
> >
> > **Returns** list of SQL statements

Compares the unique constraint to an input unique constraint and generates SQL statements to transform it into the one represented by the input.

Constraint Dictionary

Class *ConstraintDict* is a dictionary derived from *DbObjectDict* and represents the collection of constraints in a database.

class pyrseas.dbobject.constraint.**ConstraintDict**(*dbconn=None*)
> The collection of table or column constraints in a database

ConstraintDict.**from_map**(*table*, *inconstrs*, *target=''*)
> Initialize the dictionary of constraints by converting the input map
>
> > **Parameters**
> >
> > - **table** – table affected by the constraints
> > - **inconstrs** – YAML map defining the constraints
> > - **target** – column or domain indicator

4.5.4 Indexes

The index module defines two classes, *Index* and *IndexDict*, derived from DbSchemaObject and DbObjectDict, respectively.

Index

Class *Index* is derived from *DbSchemaObject* and represents an index on a database table, other than a primary key or unique constraint index. Its `keylist` attributes are the schema name, the table name and the index name. Note that index names are supposed to be unique with a given schema so the table name doesn't have to be part of the `keylist`, but has been retained to facilitate certain operations.

class `pyrseas.dbobject.index.`**`Index`**(*name*, *schema*, *table*, *description*, *unique=False*, *access_method='btree'*, *keys=[]*, *predicate=None*, *tablespace=None*, *cluster=False*, *keyexprs=None*, *defn=None*, *oid=None*)

    A physical index definition, other than a primary key or unique constraint index.

    An index is identified by its schema name and index name. However, at this time, Pyrseas uses the triple schema-table-index names as the identifier.

`Index.`**`key_expressions`**()

    Return comma-separated list of key column names and qualifiers

        Returns string

`Index.`**`to_map`**(*db*)

    Convert an index definition to a YAML-suitable format

        Returns dictionary

`Index.`**`create`**(**args*, ***kwargs*)

    Return a SQL statement to CREATE the index

        Returns SQL statements

`Index.`**`alter`**(*inindex*)

    Generate SQL to transform an existing index

        Parameters **`inindex`** – a YAML map defining the new index

        Returns list of SQL statements

    Compares the index to an input index and generates SQL statements to transform it into the one represented by the input.

`Index.`**`drop`**()

    Generate SQL to drop the current index

        Returns list of SQL statements

Index Dictionary

Class *IndexDict* is derived from *DbObjectDict* and represents the collection of indexes in a database.

class `pyrseas.dbobject.index.`**`IndexDict`**(*dbconn=None*)

    The collection of indexes on tables in a database

`IndexDict.`**`from_map`**(*table*, *inindexes*)

    Initialize the dictionary of indexes by converting the input map

        Parameters

            • **`table`** – table owning the indexes

            • **`inindexes`** – YAML map defining the indexes

4.5.5 Rules

The `rule` module defines two classes, *Rule* and *RuleDict*, derived from `DbSchemaObject` and `DbObjectDict`, respectively.

Rule

Rule is derived from *DbSchemaObject* and represents a Postgres rewrite rule.

class `pyrseas.dbobject.rule.`**Rule**(*name*, *schema*, *table*, *description*, *event*, *instead=False*, *actions=None*, *condition=None*, *definition=None*, *oid=None*)

> A rewrite rule definition

`Rule.`**identifier**()

> Return a full identifier for a rule object
>
> > **Returns** string

`Rule.`**to_map**(*db*)

> Convert rule to a YAML-suitable format
>
> > **Returns** dictionary

`Rule.`**create**(**args*, ***kwargs*)

> Return SQL statements to CREATE the rule
>
> > **Returns** SQL statements

Rule Dictionary

RuleDict is derived from *DbObjectDict*. It is a dictionary that represents the collection of rewrite rules in a database.

class `pyrseas.dbobject.rule.`**RuleDict**(*dbconn=None*)

> The collection of rewrite rules in a database.

`RuleDict.`**from_map**(*table*, *inmap*)

> Initialize the dictionary of rules by examining the input map
>
> > **Parameters**
> >
> > - **table** – the input YAML map of the associated table
> >
> > - **inmap** – the input YAML map defining the rules

4.6 Functions, Operators and Triggers

4.6.1 Functions

The `function` module defines four classes: class *Proc* derived from `DbSchemaObject`, classes *Function* and *Aggregate* derived from *Proc*, and class *ProcDict* derived from `DbObjectDict`.

Procedure

Class *Proc* is derived from *DbSchemaObject* and represents a regular or aggregate function.

class pyrseas.dbobject.function.**Proc**(*name*, *schema*, *description*, *owner*, *privileges*, *arguments*)

> A procedure such as a FUNCTION or an AGGREGATE

Proc.**extern_key**()

> Return the key to be used in external maps for this function

>> **Returns** string

Proc.**identifier**()

> Return a full identifier for a function object

>> **Returns** string

Function

Function is derived from *Proc* and represents a Postgres user-defined function.

class pyrseas.dbobject.function.**Function**(*name*, *schema*, *description*, *owner*, *privileges*, *arguments*, *language*, *returns*, *source*, *obj_file=None*, *configuration=None*, *volatility=None*, *leakproof=False*, *strict=False*, *security_definer=False*, *cost=0*, *rows=0*, *allargs=None*, *oid=None*)

> A procedural language function

Function.**to_map**(*db*, *no_owner*, *no_privs*)

> Convert a function to a YAML-suitable format

>> **Parameters**

>>> • **no_owner** – exclude function owner information

>>> • **no_privs** – exclude privilege information

>> **Returns** dictionary

Function.**create**(**args*, ***kwargs*)

> Return SQL statements to CREATE or REPLACE the function

>> **Parameters** **newsrc** – new source for a changed function

>> **Returns** SQL statements

Function.**alter**(*infunction*, *dbversion=None*, *no_owner=False*)

> Generate SQL to transform an existing function

>> **Parameters** **infunction** – a YAML map defining the new function

>> **Returns** list of SQL statements

> Compares the function to an input function and generates SQL statements to transform it into the one represented by the input.

Function.**drop**()

> Generate SQL to drop the current function

>> **Returns** list of SQL statements

Aggregate Function

Aggregate is derived from *Proc* and represents a Postgres user-defined aggregate function.

class pyrseas.dbobject.function.**Aggregate**(*name*, *schema*, *description*, *owner*, *privileges*, *arguments*, *sfunc*, *stype*, *sspace=0*, *finalfunc=None*, *finalfunc_extra=False*, *initcond=None*, *sortop=None*, *msfunc=None*, *minvfunc=None*, *mstype=None*, *msspace=0*, *mfinalfunc=None*, *mfinalfunc_extra=False*, *minitcond=None*, *kind='normal'*, *combinefunc=None*, *serialfunc=None*, *deserialfunc=None*, *parallel='unsafe'*, *oid=None*)

> An aggregate function

Aggregate.**to_map**(*db*, *no_owner*, *no_privs*)
> Convert an agggregate to a YAML-suitable format

> > **Parameters**

> > > • **no_owner** – exclude aggregate owner information

> > > • **no_privs** – exclude privilege information

> > **Returns** dictionary

Aggregate.**create**(**args*, ***kwargs*)
> Return SQL statements to CREATE the aggregate

> > **Parameters dbversion** – Posgres version

> > **Returns** SQL statements

Procedure Dictionary

ProcDict is derived from *DbObjectDict*. It is a dictionary that represents the collection of regular and aggregate functions in a database.

class pyrseas.dbobject.function.**ProcDict**(*dbconn=None*)
> The collection of regular and aggregate functions in a database

ProcDict.**from_map**(*schema*, *infuncs*)
> Initalize the dictionary of functions by converting the input map

> > **Parameters**

> > > • **schema** – schema owning the functions

> > > • **infuncs** – YAML map defining the functions

4.6.2 Operators

The operator module defines two classes: class *Operator* derived from DbSchemaObject and class *OperatorDict* derived from DbObjectDict.

Operator

Operator is derived from DbSchemaObject and represents a Postgres user-defined operator.

class pyrseas.dbobject.operator.**Operator**(*name*, *schema*, *description*, *owner*, *procedure*, *leftarg=None*, *rightarg=None*, *commutator=None*, *negator=None*, *restrict=None*, *join=None*, *hashes=False*, *merges=False*, *oid=None*)

> An operator

Operator.**extern_key**()
> Return the key to be used in external maps for this operator
>
> > **Returns** string

Operator.**qualname**()
> Return the schema-qualified name of the operator
>
> > **Returns** string
>
> No qualification is used if the schema is 'public'.

Operator.**identifier**()
> Return a full identifier for an operator object
>
> > **Returns** string

Operator.**create**(**args*, ***kwargs*)
> Return SQL statements to CREATE or REPLACE the operator
>
> > **Returns** SQL statements

Operator Dictionary

OperatorDict is derived from *DbObjectDict*. It is a dictionary that represents the collection of operators in a database.

class pyrseas.dbobject.operator.**OperatorDict**(*dbconn=None*)
> The collection of operators in a database

OperatorDict.**from_map**(*schema*, *inopers*)
> Initalize the dictionary of operators by converting the input map
>
> > **Parameters**
> >
> > - **schema** – schema owning the operators
> >
> > - **inopers** – YAML map defining the operators

4.6.3 Operator Families

The operfamily module defines two classes: class *OperatorFamily* derived from DbSchemaObject and class *OperatorFamilyDict* derived from DbObjectDict.

Operator Family

OperatorFamily is derived from DbSchemaObject and represents a Postgres operator family, a grouping of related operator classes.

class pyrseas.dbobject.operfamily.**OperatorFamily**(*name*, *schema*, *index_method*, *description*, *owner*, *oid=None*)
> An operator family

OperatorFamily.**extern_key**()
> Return the key to be used in external maps for the operator family

> > **Returns** string

OperatorFamily.**identifier**()
> Return a full identifier for an operator family object

> > **Returns** string

OperatorFamily.**create**(**args*, ***kwargs*)
> Return SQL statements to CREATE the operator family

> > **Returns** SQL statements

Operator Family Dictionary

OperatorFamilyDict is derived from *DbObjectDict*. It is a dictionary that represents the collection of operator families in a database.

class pyrseas.dbobject.operfamily.**OperatorFamilyDict**(*dbconn=None*)
> The collection of operator families in a database

OperatorFamilyDict.**from_map**(*schema*, *inopfams*)
> Initalize the dict of operator families by converting the input map

> > **Parameters**

> > > • **schema** – schema owning the operators

> > > • **inopfams** – YAML map defining the operator families

4.6.4 Operator Classes

The operclass module defines two classes: class *OperatorClass* derived from DbSchemaObject and class *OperatorClassDict* derived from DbObjectDict.

Operator Class

OperatorClass is derived from DbSchemaObject and represents a Postgres operator class.

class pyrseas.dbobject.operclass.**OperatorClass**(*name*, *schema*, *index_method*, *description*, *owner*, *family*, *type*, *default=None*, *storage=None*, *oid=None*)

> An operator class

OperatorClass.**extern_key**()
> Return the key to be used in external maps for this operator

> > **Returns** string

OperatorClass.**identifier**()
> Return a full identifier for an operator class

> > **Returns** string

OperatorClass.**to_map**(*db*, *no_owner*)
> Convert operator class to a YAML-suitable format

> > **Returns** dictionary

OperatorClass.**create**(**args*, ***kwargs*)
> Return SQL statements to CREATE the operator class
>> **Returns** SQL statements

Operator Class Dictionary

OperatorClassDict is derived from *DbObjectDict*. It is a dictionary that represents the collection of operator classes in a database.

class pyrseas.dbobject.operclass.**OperatorClassDict**(*dbconn=None*)
> The collection of operator classes in a database

OperatorClassDict.**from_map**(*schema*, *inopcls*)
> Initalize the dictionary of operator classes from the input map
>> **Parameters**
>>> - **schema** – schema owning the operator classes
>>> - **inopcls** – YAML map defining the operator classes

4.6.5 Triggers

The trigger module defines two classes, *Trigger* and *TriggerDict*, derived from DbSchemaObject and DbObjectDict, respectively.

Trigger

Trigger is derived from *DbSchemaObject* and represents a Postgres regular or constraint trigger.

class pyrseas.dbobject.trigger.**Trigger**(*name*, *schema*, *table*, *description*, *procedure*, *timing*, *level*, *events*, *constraint=False*, *deferrable=False*, *initially_deferred=False*, *columns=[]*, *condition=None*, *arguments=''*, *oid=None*)
> A procedural language trigger

Trigger.**identifier**()
> Returns a full identifier for the trigger
>> **Returns** string

Trigger.**to_map**(*db*)
> Convert a trigger to a YAML-suitable format
>> **Returns** dictionary

Trigger.**create**(**args*, ***kwargs*)
> Return SQL statements to CREATE the trigger
>> **Returns** SQL statements

Trigger Dictionary

TriggerDict is derived from *DbObjectDict*. It is a dictionary that represents the collection of triggers in a database.

---

class pyrseas.dbobject.trigger.**TriggerDict**(*dbconn=None*)
> The collection of triggers in a database

TriggerDict.**from_map**(*table*, *intriggers*)
> Initalize the dictionary of triggers by converting the input map

> **Parameters**
>> • **table** – table owning the triggers
>>
>> • **intriggers** – YAML map defining the triggers

4.7 Types and Other Schema Objects

4.7.1 Collations

The collation module defines two classes, *Collation* and *CollationDict*, derived from DbSchemaObject and DbObjectDict, respectively.

Collation

Collation is derived from *DbSchemaObject* and represents a Postgres collation.

class pyrseas.dbobject.collation.**Collation**(*name*, *schema*, *description*, *owner*, *lc_collate*, *lc_ctype*, *oid=None*)
> A collation definition

Collation.**create**(**args*, ***kwargs*)
> Return SQL statements to CREATE the collation

> **Returns** SQL statements

Collation Dictionary

CollationDict is derived from *DbObjectDict*. It is a dictionary that represents the collection of collations in a database.

class pyrseas.dbobject.collation.**CollationDict**(*dbconn=None*)
> The collection of collations in a database.

CollationDict.**from_map**(*schema*, *inmap*)
> Initalize the dictionary of collations by examining the input map

> **Parameters**
>> • **schema** – the schema owing the collations
>>
>> • **inmap** – the input YAML map defining the collations

4.7.2 Conversions

The conversion module defines two classes, *Conversion* and *ConversionDict*, derived from DbSchemaObject and DbObjectDict, respectively.

Conversion

Conversion is derived from *DbSchemaObject* and represents a Postgres conversion between character set encodings.

class pyrseas.dbobject.conversion.**Conversion**(*name*, *schema*, *description*, *owner*, *source_encoding*, *dest_encoding*, *function*, *default=False*, *oid=None*)

    A conversion definition

Conversion.**to_map**(*db*, *no_owner=False*, *no_privs=False*)

    Convert a conversion to a YAML-suitable format

        Returns dictionary

Conversion.**create**(**args*, ***kwargs*)

    Return SQL statements to CREATE the conversion

        Returns SQL statements

Conversion Dictionary

ConversionDict is derived from *DbObjectDict*. It is a dictionary that represents the collection of conversions in a database.

class pyrseas.dbobject.conversion.**ConversionDict**(*dbconn=None*)

    The collection of conversions in a database.

ConversionDict.**from_map**(*schema*, *inmap*)

    Initialize the dictionary of conversions by examining the input map

        Parameters

            • **schema** – the schema owing the conversions

            • **inmap** – the input YAML map defining the conversions

4.7.3 Text Search Objects

The textsearch module defines eight classes: classes *TSConfiguration*, *TSDictionary*, *TSParser* and *TSTemplate* derived from DbSchemaObject, and classes *TSConfigurationDict*, *TSDictionaryDict*, *TSParserDict* and *TSTemplateDict* derived from DbObjectDict.

Text Search Configuration

TSConfiguration is derived from DbSchemaObject and represents a Postgres text search configuration.

class pyrseas.dbobject.textsearch.**TSConfiguration**(*name*, *schema*, *description*, *owner*, *parser*, *oid=None*)

    A text search configuration definition

TSConfiguration.**to_map**(*db*, *no_owner*)

    Convert a text search configuration to a YAML-suitable format

        Returns dictionary

TSConfiguration.**create**(**args*, ***kwargs*)

    Return SQL statements to CREATE the configuration

> **Returns** SQL statements

Text Search Configuration Dictionary

TSConfigurationDict is derived from *DbObjectDict*. It is a dictionary that represents the collection of text search configurations in a database.

class pyrseas.dbobject.textsearch.**TSConfigurationDict**(*dbconn=None*)
> The collection of text search configurations in a database

TSConfigurationDict.**from_map**(*schema*, *inconfigs*)
> Initialize the dictionary of configs by examining the input map

>> **Parameters**

>> - **schema** – schema owning the configurations

>> - **inconfigs** – input YAML map defining the configurations

Text Search Dictionary

TSDictionary is derived from DbSchemaObject and represents a Postgres text search dictionary.

class pyrseas.dbobject.textsearch.**TSDictionary**(*name*, *schema*, *description*, *owner*, *template*, *options*, *oid=None*)
> A text search dictionary definition

TSDictionary.**create**(**args*, ***kwargs*)
> Return SQL statements to CREATE the dictionary

>> **Returns** SQL statements

Text Search Dictionary Dictionary

TSDictionaryDict is derived from *DbObjectDict*. It is a Python dictionary that represents the collection of text search dictionaries in a database.

class pyrseas.dbobject.textsearch.**TSDictionaryDict**(*dbconn=None*)
> The collection of text search dictionaries in a database

TSDictionaryDict.**from_map**(*schema*, *indicts*)
> Initialize the dictionary of dictionaries by examining the input map

>> **Parameters**

>> - **schema** – schema owning the dictionaries

>> - **indicts** – input YAML map defining the dictionaries

Text Search Parser

TSParser is derived from DbSchemaObject and represents a Postgres text search parser.

class pyrseas.dbobject.textsearch.**TSParser**(*name*, *schema*, *description*, *start*, *gettoken*, *end*, *headline*, *lextypes*, *oid=None*)
> A text search parser definition

TSParser.**create**(**args*, ***kwargs*)
> Return SQL statements to CREATE the parser

Returns SQL statements

Text Search Parser Dictionary

TSParserDict is derived from *DbObjectDict*. It is a dictionary that represents the collection of text search parsers in a database.

class pyrseas.dbobject.textsearch.**TSParserDict**(*dbconn=None*)
    The collection of text search parsers in a database

TSParserDict.**from_map**(*schema*, *inparsers*)
    Initialize the dictionary of parsers by examining the input map

    Parameters

    - **schema** – schema owning the parsers

    - **inparsers** – input YAML map defining the parsers

Text Search Template

TSTemplate is derived from DbSchemaObject and represents a Postgres text search template.

class pyrseas.dbobject.textsearch.**TSTemplate**(*name*, *schema*, *description*, *init*, *lexize*, *oid=None*)
    A text search template definition

TSTemplate.**create**(**args*, ***kwargs*)
    Return SQL statements to CREATE the template

    Returns SQL statements

Text Search Template Dictionary

TSTemplateDict is derived from *DbObjectDict*. It is a dictionary that represents the collection of text search templates in a database.

class pyrseas.dbobject.textsearch.**TSTemplateDict**(*dbconn=None*)
    The collection of text search templates in a database

TSTemplateDict.**from_map**(*schema*, *intemplates*)
    Initialize the dictionary of templates by examining the input map

    Parameters

    - **schema** – schema owning the templates

    - **intemplates** – input YAML map defining the templates

4.7.4 Types and Domains

The dbtype module defines seven classes, *DbType* derived from DbSchemaObject, *BaseType*, *Composite*, *Enum*, *Domain* and *Range* derived from *DbType*, and *TypeDict* derived from and DbObjectDict.

Database Type

Class *DbType* is derived from *DbSchemaObject* and represents a SQL type or domain as defined in the Postgres *pg_type* catalog.

class pyrseas.dbobject.dbtype.**DbType**(*name*, *schema*, *description*, *owner*, *privileges*)
> A user-defined type, such as a composite, domain or enum

Base Type

BaseType is derived from DbType and represents a Postgres user-defined base type.

The map returned by to_map() and expected as argument by *TypeDict.from_map()* has the following structure (not all fields need be present):

```
{'type t1':
    {'alignment': 'double',
     'analyze': 'analyze_func',
     'category': 'U',
     'delimiter': ',',
     'input': 'input_func',
     'internallength': 'variable',
     'output': 'output_func',
     'preferred': 'true',
     'receive': 'receive_func',
     'send': 'send_func',
     'storage': 'plain'
     'typmod_in': 'typmod_in_func',
     'typmod_out': 'typmod_out_func'
    }
}
```

class pyrseas.dbobject.dbtype.**BaseType**(*name*, *schema*, *description*, *owner*, *privileges*, *input*, *output*, *receive=None*, *send=None*, *typmod_in=None*, *typmod_out=None*, *analyze=None*, *internallength=1*, *alignment=None*, *storage=None*, *delimiter=', '*, *category=None*, *preferred=False*, *oid=None*)
> A base type

BaseType.**to_map**(*db*, *no_owner*, *no_privs*)
> Convert a type to a YAML-suitable format

> > **Parameters** **no_owner** – exclude type owner information

> > **Returns** dictionary

BaseType.**create**(**args*, ***kwargs*)
> Return SQL statements to CREATE the base type

> > **Returns** SQL statements

BaseType.**drop**()
> Generate SQL to drop the type (and related functions)

> > **Returns** list of SQL statements

Composite

Composite is derived from `DbType` and represents a standalone composite type.

class pyrseas.dbobject.dbtype.**Composite**(*name*, *schema*, *description*, *owner*, *privileges*,
oid=None)

A composite type

Composite.**to_map**(*db*, *no_owner*, *no_privs*)
Convert a type to a YAML-suitable format

Parameters no_owner – exclude type owner information

Returns dictionary

Composite.**create**(**args*, ***kwargs*)
Return SQL statements to CREATE the composite type

Returns SQL statements

Composite.**alter**(*intype*)
Generate SQL to transform an existing composite type

Parameters intype – the new composite type

Returns list of SQL statements

Compares the type to an input type and generates SQL statements to transform it into the one represented by the
input.

Enum

Enum is derived from `DbType` and represents an enumerated type.

class pyrseas.dbobject.dbtype.**Enum**(*name*, *schema*, *description*, *owner*, *privileges*, *labels*,
oid=None)

An enumerated type definition

Enum.**create**(**args*, ***kwargs*)
Return SQL statements to CREATE the enum

Returns SQL statements

Domain

Domain is derived from `DbType` and represents a domain.

class pyrseas.dbobject.dbtype.**Domain**(*name*, *schema*, *description*, *owner*, *privileges*, *type*,
not_null=False, *default=None*, *oid=None*)

A domain definition

Domain.**to_map**(*db*, *no_owner*, *no_privs*)
Convert a domain to a YAML-suitable format

Parameters no_owner – exclude domain owner information

Returns dictionary

Domain.**create**(**args*, ***kwargs*)
Return SQL statements to CREATE the domain

Returns SQL statements

Range

Range is derived from DbType and represents a Postgres range type.

class pyrseas.dbobject.dbtype.**Range**(*name*, *schema*, *description*, *owner*, *privileges*, *subtype*, *canonical=None*, *subtype_diff=None*, *oid=None*)

> A range type definition

Range.**to_map**(*db*, *no_owner*, *no_privs*)

> Convert a range type to a YAML-suitable format
>
> > **Parameters no_owner** – exclude type owner information
> >
> > **Returns** dictionary

Range.**create**(**args*, ***kwargs*)

> Return SQL statements to CREATE the range
>
> > **Returns** SQL statements

Range.**alter**(*intype*, *no_owner=False*)

> Generate SQL to transform an existing range type
>
> > **Parameters intype** – the new range type
> >
> > **Returns** list of SQL statements

> Compares the range to an input range and generates SQL statements to transform it into the one represented by the input.

Type Dictionary

TypeDict is derived from *DbObjectDict*. It is a dictionary that represents the collection of domains and enums in a database.

class pyrseas.dbobject.dbtype.**TypeDict**(*dbconn=None*)

> The collection of user-defined types in a database

TypeDict.**from_map**(*schema*, *inobjs*, *newdb*)

> Initalize the dictionary of types by converting the input map
>
> > **Parameters**
> >
> > - **schema** – schema owning the types
> > - **inobjs** – YAML map defining the schema objects
> > - **newdb** – collection of dictionaries defining the database

Augmenter API Reference

5.1 Augmenter Databases

The `augmentdb` module defines the class *AugmentDatabase*.

5.1.1 Augmenter Database

An *AugmentDatabase* is derived from *Database*. It contains two "dictionary" objects.

One is the `Dicts` container from its parent class. The *db* Dicts object, defines the database schemas, including their tables and other objects, by querying the system catalogs.

The second container is an `AugDicts` object. The *adb* AugDicts object specifies the schemas to be augmented and the augmenter configuration objects. The latter objects may be supplied either by other Augmenter modules or from the `augmenter` configuration tree on the *aug_map* supplied to the `apply()` method.

class pyrseas.augmentdb.**AugmentDatabase**(*config*)
> A database that is to be augmented

AugmentDatabase.**apply**(*aug_map*)
> Apply augmentations to an existing database

> > **Parameters aug_map** – a YAML map defining the desired augmentations

> Merges an existing database definition, as fetched from the catalogs, with an input YAML defining augmentations on various objects and an optional configuration map or the predefined configuration.

AugmentDatabase.**from_augmap**(*aug_map*)
> Populate the augment objects from the input augment map

> > **Parameters aug_map** – a YAML map defining the desired augmentations

> The *adb* holder is populated by various DbAugmentDict-derived classes by traversing the YAML augmentation map. The objects in the dictionary are then linked to related objects, e.g., tables are linked to the schemas they belong.

5.2 Augmenter Configuration Objects

These configuration objects are predefined in the Augmenter modules or can be defined or overriden by configuration elements in the `augmenter` map. Please see also *Configuration Items* and *Predefined Database Augmentations*.

5.2.1 Configuration Functions

A *CfgFunction* class specifies a Postgres function to be used by other augmenter objects. For example, this includes procedures to be invoked by triggers used to maintain audit columns. The *CfgFunctionDict* class holds all the *CfgFunction* objects, indexed by the function name and its arguments. A *CfgFunctionSource* class represents the source code for a function or part of that source code. A *CfgFunctionTemplate* class represents the source code for a function, which may include other elements that can be substituted in the final result. The class *CfgFunctionSourceDict* holds all the templates currently defined.

class pyrseas.augment.function.**CfgFunction**(***attrs*)
> A configuration function definition

CfgFunction.**apply**(*schema*, *trans_tbl*, *augdb*)
> Add a function to a given schema.

> > **Parameters**

> > > • **schema** – name of the schema in which to create the function

> > > • **trans_tbl** – translation table

> > > • **augdb** – augmenter dictionaries

class pyrseas.augment.function.**CfgFunctionDict**(*config*)
> The collection of configuration functions

CfgFunctionDict.**from_map**(*infuncs*)
> Initialize the dictionary of functions by converting the input list

> > **Parameters** **infuncs** – YAML list defining the functions

class pyrseas.augment.function.**CfgFunctionSource**(***attrs*)
> A configuration function source or part thereof

class pyrseas.augment.function.**CfgFunctionTemplate**(***attrs*)
> A configuration function source template

class pyrseas.augment.function.**CfgFunctionSourceDict**(*cfg_templates*)

5.2.2 Configuration Columns

A *CfgColumn* class defines a column to be added to a table by other augmenter objects. For example, this includes various columns that serve to capture audit trail information. The columns can be combined in various ways by the CfgAuditColumn objects. The *CfgColumnDict* class holds all the *CfgColumn* objects, indexed by column name.

class pyrseas.augment.column.**CfgColumn**(***attrs*)
> A configuration column definition

CfgColumn.**apply**(*table*)
> Add columns to the table passed in.

> > **Parameters** **table** – table to which the columns will be added

class pyrseas.augment.column.**CfgColumnDict**(*config*)
    The collection of configuration columns

CfgColumnDict.**from_map**(*incols*)
    Initialize the dictionary of columns by converting the input dict

        Parameters **incols** – YAML dictionary defining the columns

5.2.3 Configuration Triggers

A *CfgTrigger* class defines a trigger to be added to a table by other augmentation objects. For example, this includes triggers to maintain audit trail columns. The *CfgTriggerDict* class holds all the *CfgTrigger* objects, indexed by trigger name.

class pyrseas.augment.trigger.**CfgTrigger**(***attrs*)
    A configuration trigger definition

CfgTrigger.**apply**(*table*)
    Create a trigger for the table passed in.

        Parameters **table** – table on which the trigger will be created

class pyrseas.augment.trigger.**CfgTriggerDict**(*config*)
    The collection of configuration triggers

CfgTriggerDict.**from_map**(*intrigs*)
    Initialize the dictionary of triggers by converting the input dict

        Parameters **intrigs** – YAML dictionary defining the triggers

5.2.4 Configuration Audit Columns

A *CfgAuditColumn* class defines a set of attributes (columns, triggers) to be added to a table. The *CfgAuditColumnDict* class holds all the *CfgAuditColumn* objects, indexed by augmentation name.

class pyrseas.augment.audit.**CfgAuditColumn**(***attrs*)
    An augmentation that adds automatically maintained audit columns

CfgAuditColumn.**apply**(*table*, *augdb*)
    Apply audit columns to argument table.

        Parameters

            • **table** – table to which columns/triggers will be added

            • **augdb** – augment dictionaries

class pyrseas.augment.audit.**CfgAuditColumnDict**(*config*)
    The collection of audit column augmentations

CfgAuditColumnDict.**from_map**(*inaudcols*)
    Initalize the dictionary of functions by converting the input map

        Parameters **inaudcols** – YAML map defining the audit column configuration

5.3 Augmentation Objects

These objects are defined in the *aug_map* argument to the *apply* method of *AugmentDatabase*. They tie the desired augmentations, e.g., audit columns, to the tables to be affected, and the schemas owning the tables.

5.3.1 Augmentation Schema

class pyrseas.augment.schema.**AugSchema**(***attrs*)
>   A database schema definition, i.e., a named collection of tables, views, triggers and other schema objects.

AugSchema.**apply**(*augdb*)
>   Augment objects in a schema.

>   >   **Parameters augdb** – the augmenter dictionaries

class pyrseas.augment.schema.**AugSchemaDict**
>   The collection of schemas in a database

AugSchemaDict.**from_map**(*augmap*, *augdb*)
>   Initialize the dictionary of schemas by converting the augmenter map

>   >   **Parameters**

>   >   >   • **augmap** – the input YAML map defining the augmentations

>   >   >   • **augdb** – collection of dictionaries defining the augmentations

>   Starts the recursive analysis of the input map and construction of the internal collection of dictionaries describing the database objects.

AugSchemaDict.**link_current**(*schemas*)
>   Connect schemas to be augmented to actual database schemas

>   >   **Parameters schemas** – schemas in current database

AugSchemaDict.**link_refs**(*dbtables*)
>   Connect tables and functions to their respective schemas

>   >   **Parameters dbtables** – dictionary of tables

>   Fills in the *tables* dictionary for each schema by traversing the *dbtables* dictionary.

5.3.2 Augmentation Table

class pyrseas.augment.table.**AugDbClass**(***attrs*)
>   A table, sequence or view

class pyrseas.augment.table.**AugTable**(***attrs*)
>   A database table definition

AugTable.**apply**(*augdb*)
>   Augment tables in a schema.

>   >   **Parameters augdb** – the augmenter dictionaries

class pyrseas.augment.table.**AugClassDict**
>   The collection of tables and similar objects in a database

AugClassDict.**from_map**(*schema*, *inobjs*, *augdb*)
>   Initalize the dictionary of tables by converting the input map

>   >   **Parameters**

>   >   >   • **schema** – schema owning the tables

>   >   >   • **inobjs** – YAML map defining the schema objects

>   >   >   • **augdb** – collection of dictionaries defining the augmentations

`AugClassDict.`**`link_current`**(*tables*)
> Connect tables to be augmented to actual database tables

> > **Parameters** **`tables`** – tables in current schema

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

p

Symbols

A