

---

# **Pyrseas Documentation**

*Release 0.8.dev*

**Joe Abbate**

**May 22, 2017**



---

# Contents

---

<b>1</b>	<b>Features</b>	<b>3</b>
<b>2</b>	<b>Requirements</b>	<b>5</b>
<b>3</b>	<b>Contents</b>	<b>7</b>
3.1	Overview . . . . .	7
3.2	Installation . . . . .	8
3.3	Configuration . . . . .	11
3.4	Configuration Items . . . . .	12
3.5	Development . . . . .	14
3.6	Testing . . . . .	15
3.7	Known Issues . . . . .	18
3.8	Predefined Database Augmentations . . . . .	19
3.9	dbaugment - Augment a database . . . . .	20
3.10	dbtoyaml - Database to YAML . . . . .	22
3.11	yamltodb - YAML to Database . . . . .	26
3.12	Common Command Line Options . . . . .	28
<b>4</b>	<b>API Reference</b>	<b>31</b>
4.1	Database Objects . . . . .	31
4.2	Databases . . . . .	35
4.3	Casts . . . . .	36
4.4	Procedural Languages . . . . .	37
4.5	Schemas . . . . .	38
4.6	Collations . . . . .	40
4.7	Conversions . . . . .	41
4.8	Event Triggers . . . . .	41
4.9	Extensions . . . . .	42
4.10	Functions . . . . .	43
4.11	Operators . . . . .	45
4.12	Operator Families . . . . .	46
4.13	Operator Classes . . . . .	47
4.14	Types and Domains . . . . .	48
4.15	Tables, Views and Sequences . . . . .	51
4.16	Columns . . . . .	55
4.17	Constraints . . . . .	57
4.18	Indexes . . . . .	60

4.19	Rules . . . . .	61
4.20	Triggers . . . . .	62
4.21	Text Search Objects . . . . .	63
4.22	Foreign Data Objects . . . . .	66
<b>5</b>	<b>Augmenter API Reference</b>	<b>73</b>
5.1	Augmenter Databases . . . . .	73
5.2	Augmenter Configuration Objects . . . . .	74
5.3	Augmentation Objects . . . . .	75
<b>6</b>	<b>Indices and tables</b>	<b>79</b>
	<b>Python Module Index</b>	<b>81</b>

Pyrseas provides a framework and utilities to upgrade and maintain a PostgreSQL database.



# 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 will match an input YAML/JSON specification
- Generates an augmented YAML description of a PostgreSQL database from its catalogs and an augmentation specification.
- (planned) Generates a flexible web application to update PostgreSQL tables





## CHAPTER 2

---

### Requirements

---

- PostgreSQL 9.2 or higher
- Python 2.7 or higher
- Psycopg2 2.5 or higher
- PyYAML 3.10 or higher
- PgDbConn 0.8 or higher



### Overview

Pyrseas provides a framework and utilities to create, upgrade and maintain a PostgreSQL database. 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 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.

Andromeda also uses the YAML specification to generate a PHP-based application to maintain the database tables. Pyrseas *dbappgen* utility will allow a secondary YAML spec to generate a Python-based administrative application for database maintenance, which can be activated using *dbapprun*.

### Use Cases

The following two sections discuss the main scenarios where Pyrseas tools may be helpful. The first deals with the problem of controlling database structural changes while the second examines the topic of repetitive database maintenance operations.

### 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.

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 centralized or distributed 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.

## Data Maintenance

Pyrseas data administration tools (to be developed) aim to supplement the agile database development process mentioned above. While there are tools such as [pgAdmin III](#) that can be used for routine data entry tasks, their scope of action is usually a single table. For example, if you're entering data for a customer invoice, you need to know (or find by querying) the customer ID. On the other hand, [Django's admin site application](#) can present more than one table on a web page, but it requires defining the database "model" in Python and has limitations on how the database can be structured.

## Naming

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

## Installation

### 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
```

### Requirements

Pyrseas provides tools for [PostgreSQL](#), so obviously you need **PostgreSQL** to start with. Pyrseas has been tested with PG 9.2, 9.3, 9.4 and 9.5, and we'll certainly keep up with future releases. Please refer to section III, [Server Administration](#) of the PostgreSQL documentation for details on installation, setup and the various Linux, Unix and Windows platforms supported.

You will also need **Python**. Pyrseas has been tested with [Python 2.7](#), but should also work with 2.6. It has also been ported to Python 3 and tested against versions from 3.2 through 3.5. 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.

---

<sup>1</sup> The common English name for Perseas is Perseus and the Ancient Greek name is Perseos. However, in modern Greek Περσεα is the more common spelling for the mythical hero.

<sup>2</sup> He is better known for having killed Medusa.

Pyrseas talks to the PostgreSQL DBMS via the **Psycopg2 adapter**. Pyrseas has been tested with `psycopg2` 2.2 and 2.4. `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 will need to have installed the PostgreSQL and Python development packages, and a C compiler, as `pip` will 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](#).

## Downloading

Pyrseas is available at the following locations:

- [Python Package Index \(PyPI\)](#)
- [PostgreSQL Extension Network \(PGXN\)](#)
- [PgFoundry](#)
- [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 either `Pip` or `Easy Install`. See [Python Installers](#) 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).

PgFoundry offers the distribution in `gzip`-compressed tar or ZIP archive format, which can be downloaded and then installed as described [below](#).

The GitHub repository holds the Pyrseas source code, tagged according to the various releases, e.g., `v0.2.0`, 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
```

## Installation

### Extracting Sources

Once you have downloaded an archive from PyPI, PGXN or PgFoundry, 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/python2.7/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:\Python27\Lib\site-packages\pyrseas`.

### Python Installers

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

```
sudo pip install Pyrseas
```

or:

```
sudo easy_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. See the *Hitchhiker’s Guide* above for further details.

---

**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` and `easy_install` 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
```

## Configuration

The `Pyrseas` utilities allow you to configure various options through a number of `YAML` specification files.

### 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.

### 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.

### 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.

## 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 this 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 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.

## 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.

## Configuration Items

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

### Augmenter

This section is used by the `dbaument` utility (see *dbaument - Augment a database*). Most of these are specified in the system configuration file delivered with Pyrseas, but can also be included or overridden 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 for 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).

## 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.

## 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
```

## Repository

This section is used by all utilities (but **dbaument** 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.

## Development

The following details the tools needed to contribute to the development of Pyirseas. If you have any doubts or questions, please subscribe to the [Pyrseas-general mailing list](#) and post a message to the list. In addition, see *Version Control* below on how to set up a GitHub account to participate in development.

## Requirements

- Git
- Python
- PostgreSQL
- Psycopg2
- PyYAML
- Tox

## 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 Pyirseas development, you'll first have to [create a GitHub account](#).

## Programming Language

To contribute to Pyirseas, you need at least one version of [Python](#). You can develop using Python 3, but since we want to continue supporting Python 2, you'll want to install Python 2.7 in addition to Python 3.4 or 3.5.

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.

## Database Installation

To participate in Pyirseas development, you'll also need one or more installations of [PostgreSQL](#), versions 9.5, 9.4, 9.3 or 9.2. 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 [PostgreSQL.org website](#). The site also includes instructions for installing from package management systems or building it from source.

To access PostgreSQL from Python, you'll 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`.

## 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 Installers](#).

To easily run the Pyirseas tests against various Python/PostgreSQL version combinations, install [Tox](#). Please refer to [Testing](#) for more information.

## Testing

The majority of Pyirseas' 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.

## Environment Variables

By default, the tests use a PostgreSQL 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 PostgreSQL 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`

## 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, tests that define dynamically loaded functions (e.g., `test_create_c_lang_function()` in `test_function.py`) require `SUPERUSER` privilege. Such tests will be skipped if the user lacks the privilege.

Most tests do not require installation of supporting PostgreSQL packages. However, tests that define dynamically loaded functions (see above) require that the `contrib/spi` module be installed.

On Windows, it is necessary to install Perl in order to run some of the tests. A suitable choice is 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 PostgreSQL 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, run under PostgreSQL 9.2 and later, require the `fr_FR.utf8` locale (or `French.France.1252` language on Windows) to be installed.

## 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 either `pip` or `easy_install`. Some operations require administrative or superuser privileges, at either the operating system or PostgreSQL level.

- Install Git using package manager or from <http://git-scm.com/download> (on Windows, prefer Git Bash)
- `git clone git://github.com/perseas/Pyrseas.git`
- Install Python 2.7 and 3.4 or 3.5, using package manager or from installers at <http://www.python.org/download/>.

- Install PostgreSQL 9.5, 9.4, 9.3 and 9.2, using package manager or binary installers at <http://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 PostgreSQL version number)

---

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

---

**Note:** On Windows, it is best to first install the 2008 Microsoft Visual Express Studio from [here](#). An alternative that may work is to use [MinGW](#). See [these blog posts](#) for more details.

---

- Install PyYAML, using package manager, or from PyPI (<http://pypi.python.org/pypi/PyYAML/>) or <http://pyyaml.org/download/pyyaml/>.
- Install Tox, from PyPI (<http://pypi.python.org/pypi/tox>)

---

**Note:** Psycopg2, PyYAML and Tox all have to be installed twice, i.e., once under Python 2.7 and another under 3.4 or 3.5.

---

- 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 PostgreSQL password file, e.g., on Linux: `~/.pgpass`, on Windows: `%APPDATA%\postgresql\pgpass.conf`.
- Create directories to hold tablespaces, e.g., `/extra/pg/9.5/ts1` on Linux, `C:\extra\pg\9.5\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 or pgAdmin, 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).
  - On Windows, for PostgreSQL 9.2, the default installation is owned by the Network Service account, so the `cacls` command should be `cacls <dir> /E /G networkservices:F`.

---

**Note:** The creation of users/roles and tablespaces has to be repeated for each PostgreSQL version.

---

- 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 `PG92_PORT`, `PG93_PORT`, `PG94_PORT` and `PG95_PORT` to point to the corresponding PostgreSQL ports.
- Invoke `tox`. This will create two virtualenvs in a `.tox` subdirectory—one for Python 2.7 and another for 3.5 or 3.4, install Pyrseas and its prerequisites (Psychopg2 and PyYAML) into each virtualenv and run the unit tests for each combination of PostgreSQL and Python.

## Known Issues

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

### 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.

### 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.

### Multiline Strings

The text of function source code, view definitions or object `COMMENT`s 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 = \n        \ $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\n\n        SELECT inventory_id\n        FROM inventory\n        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.

## Views Dependent on Primary Key

**yamltodb** may fail to recreate a view if it takes advantage of the Postgres enhancement, introduced in version 9.1, where a `SELECT` with `GROUP BY` includes columns which are functionally dependent on the grouped columns and the latter are the primary key of the table containing the additional columns. Please refer to the [GROUP BY documentation](#) for further details.

A potential workaround, that may be used if such views are not referenced elsewhere, would be to edit the YAML specification to remove the views (either all or those affected by this) from a first pass through **yamltodb**. This should create the tables with their primary keys and then a second pass will create the views.

## Predefined Database Augmentations

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

### 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.

### 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 web application 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 correctly.

## 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_aud_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`.

## dbaugment - Augment a database

### Name

`dbaugment` – Augment a PostgreSQL database in predefined ways

### Synopsis

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



## Description

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

The specification file format is as follows:

```

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 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*).

**dbaument** first reads the database catalogs. It also initializes itself from predefined configuration information. **dbaument** 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.

## Options

**dbaument** 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 augmented.

### spec

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

## Examples

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

```
dbaument moviesdb moviesbl.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
```

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

```
dbaument moviesdb film.yaml | yamltodb moviesdb -u
```

## See Also

*Predefined Database Augmentations*

## dbtoyaml - Database to YAML

### Name

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

### Synopsis

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

### Description

**dbtoyaml** is a utility for extracting the schema of a PostgreSQL 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, 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.

The output format is as follows:

```
schema public:
  owner: postgres
  privileges:
  - postgres:
    - all
  - PUBLIC:
    - all
  table t1:
    check_constraints:
      check_expr: (c2 > 123)
      columns:
      - c2
    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
      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 PostgreSQL. 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 PostgreSQL database objects. See [API Reference](#) for a list of supported objects.

## 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 PostgreSQL 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 PostgreSQL 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 -d` outputs a special YAML “index” file, named `database.<dbname>.yaml` in the root directory. When `dbtoyaml -d` 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.

## 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`.

## 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 -d movies/dbspec
```

## yamltodb - YAML to Database

### Name

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

### Synopsis

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

### Description

`yamltodb` is a utility for generating SQL statements to update a PostgreSQL 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` outputs the following SQL statements:

```
CREATE SCHEMA s1;
CREATE TABLE t1 (
    c1 integer NOT NULL,
    c2 smallint,
    c3 boolean DEFAULT false,
    c4 text);
CREATE TABLE s1.t2 (
    c21 integer NOT NULL,
    c22 character varying(16));
ALTER TABLE s1.t2 ADD CONSTRAINT t2_pkey PRIMARY KEY (c21);
ALTER TABLE t1 ADD CONSTRAINT t1_pkey PRIMARY KEY (c1);
ALTER TABLE t1 ADD CONSTRAINT t1_c2_fkey FOREIGN KEY (c2) REFERENCES s1.t2 (c21);
```

### 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 be 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.

### --quote-reserved

When generating SQL, use delimited (quoted) identifiers around reserved words used as identifiers, e.g., a table named "order". Normally, only identifiers with embedded spaces or other disallowed characters are quoted.

### --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.

## 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 output 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
```

## 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 PostgreSQL server is running. The default host name is determined by PostgreSQL (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 PostgreSQL server is listening for connections. The default port number is determined by PostgreSQL (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>  
*User name* to connect as. The default user name is determined by PostgreSQL (normally, the name of the user running the program).
- version**  
Print the program version 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 `PGPASSWORD` 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
```



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.

## Database Objects

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

### Database Object

A *DBObject* represents a database object such as a schema, table, or column, defined in a PostgreSQL *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=None, description=None, owner=None, privileges=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 FDWs. 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.

`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 maps. 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 overridden methods) other elements, e.g., the arguments to a function.

`DBObject`.**to\_map** (*no\_owner=False, no\_privs=False*)

Convert an object to a YAML-suitable format

**Parameters**

- **no\_owner** – exclude object owner information
- **no\_privs** – exclude privilege information

**Returns** dictionary

This base implementation simply copies the internal Python dictionary, removes the *keylist* attributes, and returns a new dictionary using the `extern_key()` result as the key.

`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()`  
Return SQL statement to DROP the object

**Returns** SQL statement

`DBObject.rename(newname)`  
Return SQL statement to RENAME the object

**Parameters** `newname` – the new name for the object

**Returns** SQL statement

`DBObject.diff_map(inobj, no_owner=False)`  
Generate SQL to transform an existing object

**Parameters**

- `inobj` – a YAML map defining the new object
- `no_owner` – exclude object owner information

**Returns** list of SQL statements

Compares the object to an input object and generates SQL statements to transform it into the one represented by the input. This base implementation simply deals with owners and comments.

`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

## 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 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

`DbObjectDict.cls` = `<class 'pyrseas.dbobject.DbObject'>`

The class, derived from *DbObject* that the objects belong to.

`DbObjectDict.query` = ''

The SQL SELECT query to fetch object instances from the catalogs

This is used by the method *fetch()*.

`DbObjectDict.to_map` (*opts*)

Convert the object dictionary to a regular dictionary

**Parameters** *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 class *query*

**Returns** list of `self.cls` objects

## 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 itself.

**class** `pyrseas.dbobject.DbSchemaObject` (*schema, name, description=None, owner=None, privileges=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` (*objname=None*)

Return the schema-qualified name of self or a related object

**Returns** string

No qualification is used if the schema is 'public'.

`DbSchemaObject.unqualify()`

Adjust the schema and table name if the latter is qualified

`DbSchemaObject.drop()`

Return a SQL DROP statement for the schema object

**Returns** SQL statement

`DbSchemaObject.rename` (*newname*)

Return a SQL ALTER statement to RENAME the schema object

**Parameters** *newname* – the new name of the object

**Returns** SQL statement

## Databases

The database module defines *Database*.

### Database

A *Database* is initialized from a *CatDbConnection* object (a specialized class derived from *DbConnection*). It consists of one or two *Dicts*. 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*). For example, the *ClassDict* dictionary is indexed by (*schema name*, *table 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– defines the database schemas, including their tables and other objects, by querying the system catalogs. The *ndb Dicts* object defines 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 dictionary 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()**

Populate the database objects by querying the catalogs

The *db* holder is populated by various *DbObjectDict*-derived classes by querying the catalogs. 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 metadata directory

**Returns** dictionary

Database.**to\_map**()

Convert the db maps to a single hierarchy suitable for YAML

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

Database.**diff\_map**(*input\_map*)

Generate SQL to transform an existing database

**Parameters** *input\_map* – a YAML map defining the new database

**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.

## 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 PostgreSQL cast.

```
class pyrseas.dbobject.cast.Cast (name=None, description=None, owner=None, privileges=None,
                                  **attrs)
```

A cast

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**(*no\_owner=False, no\_privs=False*)

Convert a cast to a YAML-suitable format

**Returns** dictionary

Cast.**create**(*obj, \*args, \*\*kwargs*)

Return SQL statements to CREATE the cast

**Returns** SQL statements

Cast.**diff\_map**(*inobj, no\_owner=False*)

Generate SQL to transform an existing object

**Parameters**

- **inobj** – a YAML map defining the new object
- **no\_owner** – exclude object owner information



**Returns** list of SQL statements

Compares the object to an input object and generates SQL statements to transform it into the one represented by the input. This base implementation simply deals with owners and comments.

## 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*)

Initialize the dictionary of casts by converting the input map

### Parameters

- **incasts** – YAML map defining the casts
- **newdb** – collection of dictionaries defining the database

**CastDict.diff\_map** (*incasts*)

Generate SQL to transform existing casts

**Parameters** **incasts** – a YAML map defining the new casts

**Returns** list of SQL statements

Compares the existing cast definitions, as fetched from the catalogs, to the input map and generates SQL statements to transform the casts accordingly.

## 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=None, description=None, owner=None, privileges=None, \*\*attrs*)

A procedural language definition

**Language.to\_map** (*no\_owner, no\_privs*)

Convert language to a YAML-suitable format

**Parameters** **no\_owner** – exclude language owner information

**Returns** dictionary

**Language.create** ()

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** `pyrease.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

`LanguageDict.link_refs` (*dbfunctions, langs*)

Connect functions to their respective languages

**Parameters** `dbfunctions` – dictionary of functions

Fills in the *functions* dictionary for each language by traversing the *dbfunctions* dictionary, which is keyed by schema and function name.

`LanguageDict.diff_map` (*inlanguages*)

Generate SQL to transform existing languages

**Parameters** `input_map` – a YAML map defining the new languages

**Returns** list of SQL statements

Compares the existing language definitions, as fetched from the catalogs, to the input map and generates SQL statements to transform the languages accordingly.

## Schemas

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

### Schema

*Schema* is derived from *DBObject* and represents a database schema, i.e., a collection of tables and other objects. The 'public' schema is special 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."

For now, the schema name is the only attribute and is of course the identifying attribute in the *Schema* `keylist`.

**class** `pyrease.dbobject.schema.Schema` (*name=None, description=None, owner=None, privileges=None, \*\*attrs*)

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` (*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** (*obj*, \**args*, \*\**kwargs*)

Return SQL statements to CREATE the schema

**Returns** 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

## 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.dboject.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.**link\_refs** (*db*, *datacopy*)

Connect various schema objects to their respective schemas

**Parameters**

- **db** – dictionary of dictionaries of all objects
- **datacopy** – dictionary of data copying info

SchemaDict.**to\_map** (*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.**diff\_map** (*inschemas*)

Generate SQL to transform existing schemas

**Parameters** *input\_map* – a YAML map defining the new schemas

**Returns** list of SQL statements

Compares the existing schema definitions, as fetched from the catalogs, to the input map and generates SQL statements to transform the schemas accordingly.

`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

## 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 PostgreSQL collation (available on PostgreSQL 9.1 or later).

`class pyrease.dboobject.collation.Collation` (*schema, name, description=None, owner=None, privileges=None, \*\*attrs*)

A collation definition

`Collation.create` (*obj, \*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 pyrease.dboobject.collation.CollationDict` (*dbconn=None*)

The collection of collations in a database.

`CollationDict.from_map` (*schema, inmap*)

Initialize the dictionary of collations by examining the input map

**Parameters**

- **schema** – the schema owning the collations
- **inmap** – the input YAML map defining the collations

`CollationDict.diff_map` (*incolls*)

Generate SQL to transform existing collations

**Parameters** *incolls* – a YAML map defining the new collations

**Returns** list of SQL statements

Compares the existing collation definitions, as fetched from the catalogs, to the input map and generates SQL statements to create, drop or change the collations accordingly.

## 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 PostgreSQL conversion between character set encodings.

```
class pyrseas.dboobject.conversion.Conversion (schema, name, description=None,
                                             owner=None, privileges=None, **attrs)
```

A conversion definition

```
Conversion.create (obj, *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.dboobject.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 owning the conversions
- **inmap** – the input YAML map defining the conversions

```
ConversionDict.diff_map (inconvs)
```

Generate SQL to transform existing conversions

**Parameters** **inconv**s – a YAML map defining the new conversions

**Returns** list of SQL statements

Compares the existing conversion definitions, as fetched from the catalogs, to the input map and generates SQL statements to create, drop or change the conversions accordingly.

## 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 PostgreSQL 9.3 onwards.

**class** `pyrseas.dbobject.eventtrig.EventTrigger` (*name=None, description=None, owner=None, privileges=None, \*\*attrs*)

An event trigger

`EventTrigger.create` (*obj, \*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*)

Initialize the dictionary of triggers by converting the input map

### Parameters

- **intriggers** – YAML map defining the event triggers
- **newdb** – dictionary of input database

`EventTriggerDict.diff_map` (*intriggers*)

Generate SQL to transform existing event triggers

**Parameters** **intriggers** – a YAML map defining the new event triggers

**Returns** list of SQL statements

Compares the existing event trigger definitions, as fetched from the catalogs, to the input map and generates SQL statements to transform the event triggers accordingly.

## 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 PostgreSQL extension.

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

An extension

`Extension.create` (*obj, \*args, \*\*kwargs*)

Return SQL statements to CREATE the extension

**Returns** SQL statements

## Extension Dictionary

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

**class** `pyrease.dbobject.extension.ExtensionDict` (*dbconn=None*)

The collection of extensions in a database

`ExtensionDict.from_map` (*inexts, langtempl, newdb*)

Initialize the dictionary of extensions by converting the input map

### Parameters

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

`ExtensionDict.diff_map` (*inexts*)

Generate SQL to transform existing extensions

**Parameters** **inexts** – a YAML map defining the new extensions

**Returns** list of SQL statements

Compares the existing extension definitions, as fetched from the catalogs, to the input map and generates SQL statements to transform the extensions accordingly.

## Functions

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

### Procedure

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

**class** `pyrease.dbobject.function.Proc` (*schema, name, description=None, owner=None, privileges=None, \*\*attrs*)

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 PostgreSQL user-defined function.

**class** `pyrease.dbobject.function.Function` (*schema, name, description=None, owner=None, privileges=None, \*\*attrs*)

A procedural language function

Function.**to\_map** (*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** (*obj*, *\*args*, *\*\*kwargs*)

Return SQL statements to CREATE or REPLACE the function

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

**Returns** SQL statements

Function.**diff\_map** (*infunction*)

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.

## Aggregate Function

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

**class** `pyrseas.dbobject.function.Aggregate` (*schema*, *name*, *description=None*, *owner=None*, *privileges=None*, *\*\*attrs*)

An aggregate function

Aggregate.**to\_map** (*no\_owner*, *no\_privs*)

Convert an aggregate to a YAML-suitable format

**Parameters**

- **no\_owner** – exclude aggregate owner information
- **no\_privs** – exclude privilege information

**Returns** dictionary

Aggregate.**create** (*obj*, *\*args*, *\*\*kwargs*)

Return SQL statements to CREATE the aggregate

**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*)

Initialize the dictionary of functions by converting the input map



**Parameters**

- **schema** – schema owning the functions
- **infuncs** – YAML map defining the functions

`ProcDict.diff_map(infuncs)`

Generate SQL to transform existing functions

**Parameters** **infuncs** – a YAML map defining the new functions

**Returns** list of SQL statements

Compares the existing function definitions, as fetched from the catalogs, to the input map and generates SQL statements to transform the functions accordingly.

## 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 PostgreSQL user-defined operator.

**class** `pyrseas.dbobject.operator.Operator`(*schema*, *name*, *description=None*, *owner=None*, *privileges=None*, *\*\*attrs*)

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(obj, *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*)

Initialize the dictionary of operators by converting the input map

**Parameters**

- **schema** – schema owning the operators
- **inopers** – YAML map defining the operators

`OperatorDict.diff_map` (*inopers*)

Generate SQL to transform existing operators

**Parameters** **inopers** – a YAML map defining the new operators

**Returns** list of SQL statements

Compares the existing operator definitions, as fetched from the catalogs, to the input map and generates SQL statements to transform the operators accordingly.

## 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 PostgreSQL operator family.

```
class pyrseas.dboobject.operfamily.OperatorFamily (schema, name, description=None,
                                                    owner=None, privileges=None, **attrs)
```

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` (*obj*, *\*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.dboobject.operfamily.OperatorFamilyDict (dbconn=None)
```

The collection of operator families in a database

`OperatorFamilyDict.from_map` (*schema*, *inopfams*)

Initialize the dict of operator families by converting the input map

**Parameters**

- **schema** – schema owning the operators
- **inopfams** – YAML map defining the operator families

OperatorFamilyDict.**diff\_map** (*inopfams*)

Generate SQL to transform existing operator families

**Parameters** **inopfams** – a YAML map defining the new operator families

**Returns** list of SQL statements

Compares the existing operator family definitions, as fetched from the catalogs, to the input map and generates SQL statements to transform the operator families accordingly.

## 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 PostgreSQL operator class.

```
class pyrseas.dboobject.operclass.OperatorClass (schema, name, description=None,
                                                owner=None, privileges=None, **attrs)
```

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** (*no\_owner*)

Convert operator class to a YAML-suitable format

**Returns** dictionary

OperatorClass.**create** (*obj, \*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.dboobject.operclass.OperatorClassDict (dbconn=None)
```

The collection of operator classes in a database

OperatorClassDict.**from\_map** (*schema, inopcls*)

Initialize the dictionary of operator classes from the input map

**Parameters**

- **schema** – schema owning the operator classes
- **inopcls** – YAML map defining the operator classes

OperatorClassDict.**diff\_map** (*inopcls*)

Generate SQL to transform existing operator classes

**Parameters** **inopcls** – a YAML map defining the new operator classes

**Returns** list of SQL statements

Compares the existing operator class definitions, as fetched from the catalogs, to the input map and generates SQL statements to transform the operator classes accordingly.

## Types and Domains

The `dbtype` module defines six classes, `DbType` derived from `DbSchemaObject`, `BaseType`, `Composite`, `Enum` and `Domain` derived from `DbType`, and `TypeDict` derived from `DbObjectDict`.

### Database Type

Class `DbType` is derived from `DbSchemaObject` and represents a SQL type or domain as defined in the PostgreSQL `pg_type` catalog. Note: Only enumerated types are implemented currently.

```
class pyrseas.dbobject.dbtype.DbType(schema, name, description=None, owner=None, privileges=None, **attrs)
```

A composite, domain or enum type

### Base Type

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

The map returned by `to_map()` and expected as argument by `diff_map()` has the following structure (not all fields need be present):

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

```
class pyrseas.dbobject.dbtype.BaseType(schema, name, description=None, owner=None, privileges=None, **attrs)
```

A composite type

`BaseType.to_map` (*no\_owner*)

Convert a type to a YAML-suitable format

**Parameters** **no\_owner** – exclude type owner information

**Returns** dictionary

`BaseType.create(obj, *args, **kwargs)`

Return SQL statements to CREATE the base type

**Returns** SQL statements

`BaseType.drop()`

Return SQL statement to DROP the base type

**Returns** SQL statement

We have to override the super method and add CASCADE to drop dependent functions.

## Composite

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

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

A composite type

`Composite.to_map(no_owner)`

Convert a type to a YAML-suitable format

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

**Returns** dictionary

`Composite.create(obj, *args, **kwargs)`

Return SQL statements to CREATE the composite type

**Returns** SQL statements

`Composite.diff_map(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(schema, name, description=None, owner=None, privileges=None, **attrs)`

An enumerated type definition

`Enum.create(obj, *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(schema, name, description=None, owner=None, privileges=None, **attrs)
```

A domain definition

```
Domain.to_map(no_owner)
```

Convert a domain to a YAML-suitable format

**Parameters** *no\_owner* – exclude domain owner information

**Returns** dictionary

```
Domain.create(obj, *args, **kwargs)
```

Return SQL statements to CREATE the domain

**Returns** SQL statements

## 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 domains and enums in a database

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

Initialize 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

```
TypeDict.link_refs(dbcolumns, dbconstrs, dbfuncs)
```

Connect various objects to their corresponding types or domains

**Parameters**

- **dbcolumns** – dictionary of columns
- **dbconstrs** – dictionary of constraints
- **dbfuncs** – dictionary of functions

Fills the *check\_constraints* dictionaries for each domain by traversing the *dbconstrs* dictionary. Fills the attributes list for composite types. Fills the dependent functions dictionary for base types.

```
TypeDict.diff_map(intypes)
```

Generate SQL to transform existing domains and types

**Parameters** *intypes* – a YAML map defining the new domains/types

**Returns** list of SQL statements

Compares the existing domain/type definitions, as fetched from the catalogs, to the input map and generates SQL statements to transform the domains/types accordingly.

## Tables, Views and Sequences

The `table` module defines 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 PostgreSQL `pg_class` catalog.

```
class pyrseas.dboobject.table.DbClass(schema, name, description=None, owner=None, privileges=None, **attrs)
```

A table, sequence or view

### Sequence

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

A `Sequence` has the following attributes: `start_value`, `increment_by`, `max_value`, `min_value` and `cache_value`.

The map returned by `to_map()` and expected as argument by `diff_map()` has the following structure:

```
{'sequence seq1':
  {'start_value': 1,
   'increment_by': 1,
   'max_value': None,
   'min_value': None,
   'cache_value': 1
  }
}
```

Only the inner dictionary is passed to `diff_map()`. The values are defaults so in practice an empty dictionary is also acceptable.

```
class pyrseas.dboobject.table.Sequence(schema, name, description=None, owner=None, privileges=None, **attrs)
```

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(opts)`

Convert a sequence definition to a YAML-suitable format

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

**Returns** dictionary

Sequence.**create** (*obj*, \**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.**diff\_map** (*inseq*)  
 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.

## 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 *diff\_map* () has a structure similar to the following:

```
{'table t1':
  {'columns':
    [
      {'c1': {'type': 'integer', 'not_null': True}},
      {'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** `pyrease.dbobject.table.Table`(*schema, name, description=None, owner=None, privileges=None, \*\*attrs*)

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.

`Table.column_names()`

Return a list of column names in the table

**Returns** list

`Table.to_map(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()`

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.diff_map(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.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

## 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 `diff_map()` has a structure similar to the following:

```
{'view v1':
  {'definition': " SELECT ...;",
   'description': "this is the comment for view v1"
  }
}
```

**class** `pyrseas.dbobject.table.View`(*schema*, *name*, *description=None*, *owner=None*, *privileges=None*, **\*\*attrs**)

A database view definition

A view is identified by its schema name and view name.

`View.to_map`(*opts*)

Convert a view to a YAML-suitable format

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

**Returns** dictionary

`View.create`(*obj*, *\*args*, **\*\*kwargs**)

Return SQL statements to CREATE the table

**Returns** SQL statements

`View.diff_map`(*inview*)

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](#), available from PostgreSQL 9.3 onwards. Its `keylist` attributes are the schema name and the view name.

**class** `pyrseas.dbobject.table.MaterializedView`(*schema*, *name*, *description=None*, *owner=None*, *privileges=None*, **\*\*attrs**)

A materialized view definition

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

`MaterializedView.to_map`(*opts*)

Convert a materialized view to a YAML-suitable format

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

**Returns** dictionary

`MaterializedView.diff_map`(*inview*)

Generate SQL to transform an existing materialized 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.

## Class Dictionary

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

**class** `pyrease.dbobject.table.ClassDict` (*dbconn=None*)

The collection of tables and similar objects in a database

`ClassDict.from_map` (*schema, inobjs, newdb*)

Initialize 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

`ClassDict.link_refs` (*dbcolumns, dbconstrs, dbindexes, dbrules, dbtriggers*)

Connect columns, constraints, etc. to their respective tables

### Parameters

- **dbcolumns** – dictionary of columns
- **dbconstrs** – dictionary of constraints
- **dbindexes** – dictionary of indexes
- **dbrules** – dictionary of rules
- **dbtriggers** – dictionary of triggers

Links each list of table columns in *dbcolumns* to the corresponding table. Fills the *foreign\_keys*, *unique\_constraints*, *indexes* and *triggers* dictionaries for each table by traversing the *dbconstrs*, *dbindexes* and *dbtriggers* dictionaries, which are keyed by schema, table and constraint, index or trigger name.

`ClassDict.diff_map` (*intables*)

Generate SQL to transform existing tables and sequences

**Parameters** *intables* – a YAML map defining the new tables/sequences

**Returns** list of SQL statements

Compares the existing table/sequence definitions, as fetched from the catalogs, to the input map and generates SQL statements to transform the tables/sequences accordingly.

## 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 in a table, or an attribute in a composite type. Its `keylist` attributes are the schema name and the table name.

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

```
class pyrease.dbobject.column.Column(schema, name, description=None, owner=None, privileges=None, **attrs)
```

A table column definition

`Column.to_map(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.set_sequence_default()`

Return SQL statements to set a nextval() DEFAULT

**Returns** list of SQL statements

`Column.diff_map(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** `pyrease.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

`ColumnDict.diff_map` (*incols*)  
Generate SQL to transform existing columns

**Parameters** **incols** – a YAML map defining the new columns

**Returns** list of SQL statements

Compares the existing column definitions, as fetched from the catalogs, to the input map and generates SQL statements to transform the columns accordingly.

This takes care of dropping columns that are not present in the input map. It's separate so that it can be done last, after other table, constraint and index changes.

## 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** `pyrease.dbobject.constraint.Constraint` (*schema, name, description=None, owner=None, privileges=None, \*\*attrs*)  
A constraint definition, such as a primary key, foreign key or unique constraint

`Constraint.key_columns` ()  
Return comma-separated list of key column names

**Returns** string

`Constraint.add` (*obj, \*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` (*schema*, *name*, *description=None*,  
*owner=None*, *privileges=None*, **\*\*attrs**)

A check constraint definition

`CheckConstraint.to_map(dbcols)`  
 Convert a check constraint definition to a YAML-suitable format  
**Parameters** `dbcols` – dictionary of dbobject columns  
**Returns** dictionary

`CheckConstraint.add(obj, *args, **kwargs)`  
 Return string to add the CHECK constraint via ALTER TABLE  
**Returns** SQL statement

`CheckConstraint.diff_map(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` (*schema*, *name*, *description=None*,  
*owner=None*, *privileges=None*, **\*\*attrs**)

A primary key constraint definition

`PrimaryKey.to_map(dbcols)`  
 Convert a primary key definition to a YAML-suitable format  
**Parameters** `dbcols` – dictionary of dbobject columns  
**Returns** dictionary

`PrimaryKey.diff_map(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.diff\_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.dboject.constraint.ForeignKey` (*schema*, *name*, *description=None*, *owner=None*, *privileges=None*, *\*\*attrs*)

A foreign key constraint definition

`ForeignKey.ref_columns` ()

Return comma-separated list of reference column names

**Returns** string

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

Convert a foreign key definition to a YAML-suitable format

**Parameters** *dbcols* – dictionary of `dboject` columns

**Returns** dictionary

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

Return string to add the foreign key via ALTER TABLE

**Returns** SQL statement

`ForeignKey.diff_map` (*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.dboject.constraint.UniqueConstraint` (*schema*, *name*, *description=None*, *owner=None*, *privileges=None*, *\*\*attrs*)

A unique constraint definition

`UniqueConstraint.to_map` (*dbcols*)

Convert a unique constraint definition to a YAML-suitable format

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

**Returns** dictionary

`UniqueConstraint.diff_map` (*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=',, rtables=None*)

Initialize the dictionary of constraints by converting the input map

**Parameters**

- **table** – table affected by the constraints
- **inconstrs** – YAML map defining the constraints

`ConstraintDict.diff_map` (*inconstrs*)

Generate SQL to transform existing constraints

**Parameters** `inconstrs` – a YAML map defining the new constraints

**Returns** list of SQL statements

Compares the existing constraint definitions, as fetched from the catalogs, to the input map and generates SQL statements to transform the constraints accordingly.

## 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.

An `Index` has the following attributes: `access_method`, `unique`, and `keycols`.

**class** `pyrseas.dbobject.index.Index` (*schema, name, description=None, owner=None, privileges=None, \*\*attrs*)

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



`Index.key_expressions ()`

Return comma-separated list of key column names and qualifiers

**Returns** string

`Index.to_map ()`

Convert an index definition to a YAML-suitable format

**Returns** dictionary

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

Return a SQL statement to CREATE the index

**Returns** SQL statements

`Index.diff_map (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 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

`IndexDict.diff_map (inindexes)`

Generate SQL to transform existing indexes

**Parameters** `inindexes` – a YAML map defining the new indexes

**Returns** list of SQL statements

Compares the existing index definitions, as fetched from the catalogs, to the input map and generates SQL statements to transform the indexes accordingly.

## 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 PostgreSQL rewrite rule.

```
class pyrease.dbobject.rule.Rule(schema, name, description=None, owner=None, privileges=None, **attrs)
```

A rewrite rule definition

```
Rule.identified()
```

Return a full identifier for a rule object

**Returns** string

```
Rule.to_map()
```

Convert rule to a YAML-suitable format

**Returns** dictionary

```
Rule.create(obj, *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 pyrease.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** *inmap* – the input YAML map defining the rules

```
RuleDict.diff_map(inrules)
```

Generate SQL to transform existing rules

**Parameters** *input\_map* – a YAML map defining the new rules

**Returns** list of SQL statements

Compares the existing rule definitions, as fetched from the catalogs, to the input map and generates SQL statements to transform the rules accordingly.

## 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 PostgreSQL regular trigger or constraint trigger.

```
class pyrease.dbobject.trigger.Trigger(schema, name, description=None, owner=None, privileges=None, **attrs)
```

A procedural language trigger

`Trigger.identifier()`  
Returns a full identifier for the trigger

**Returns** string

`Trigger.to_map()`  
Convert a trigger to a YAML-suitable format

**Returns** dictionary

`Trigger.create(obj, *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)`  
Initialize the dictionary of triggers by converting the input map

### Parameters

- **table** – table owning the triggers
- **intriggers** – YAML map defining the triggers

`TriggerDict.diff_map(intriggers)`  
Generate SQL to transform existing triggers

**Parameters** `intriggers` – a YAML map defining the new triggers

**Returns** list of SQL statements

Compares the existing trigger definitions, as fetched from the catalogs, to the input map and generates SQL statements to transform the triggers accordingly.

## 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 PostgreSQL text search configuration.

`class pyrseas.dbobject.textsearch.TSConfiguration(schema, name, description=None, owner=None, privileges=None, **attrs)`

A text search configuration definition

`TSConfiguration.to_map(no_owner)`  
Convert a text search configuration to a YAML-suitable format

**Returns** dictionary

`TSConfiguration.create(obj, *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.dboobject.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

`TSConfigurationDict.diff_map(inconfigs)`

Generate SQL to transform existing configurations

**Parameters** `input_map` – a YAML map defining the new configurations

**Returns** list of SQL statements

Compares the existing configuration definitions, as fetched from the catalogs, to the input map and generates SQL statements to transform the configurations accordingly.

## Text Search Dictionary

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

**class** `pyrseas.dboobject.textsearch.TSDictionary` (*schema, name, description=None, owner=None, privileges=None, \*\*attrs*)

A text search dictionary definition

`TSDictionary.create(obj, *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.dboobject.textsearch.TSDictionaryDict` (*dbconn=None*)

The collection of text search dictionaries in a database

`TSDictionaryDict.from_map(schema, indict)`

Initialize the dictionary of dictionaries by examining the input map

### Parameters

- **schema** – schema owning the dictionaries

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

`TSDictionaryDict.diff_map(indicts)`

Generate SQL to transform existing dictionaries

**Parameters** `input_map` – a YAML map defining the new dictionaries

**Returns** list of SQL statements

Compares the existing dictionary definitions, as fetched from the catalogs, to the input map and generates SQL statements to transform the dictionaries accordingly.

## Text Search Parser

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

**class** `pyrease.dboject.textsearch.TSParser` (`schema`, `name`, `description=None`, `owner=None`, `privileges=None`, `**attrs`)

A text search parser definition

`TSParser.create(obj, *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** `pyrease.dboject.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

`TSParserDict.diff_map(inparsers)`

Generate SQL to transform existing parsers

**Parameters** `input_map` – a YAML map defining the new parsers

**Returns** list of SQL statements

Compares the existing parser definitions, as fetched from the catalogs, to the input map and generates SQL statements to transform the parsers accordingly.

## Text Search Template

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

**class** `pyrease.dboject.textsearch.TSTemplate` (`schema`, `name`, `description=None`, `owner=None`, `privileges=None`, `**attrs`)

A text search template definition

`TSTemplate.create(obj, *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

`TSTemplateDict.diff_map(intemplates)`  
 Generate SQL to transform existing templates

**Parameters** `input_map` – a YAML map defining the new templates

**Returns** list of SQL statements

Compares the existing template definitions, as fetched from the catalogs, to the input map and generates SQL statements to transform the templates accordingly.

## Foreign Data Objects

The `foreign` module defines nine classes: `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 function dealing with the `OPTIONS` clauses common to the foreign data objects.

**class** `pyrseas.dbobject.foreign.DbObjectWithOptions(name=None, description=None, owner=None, privileges=None, **attrs)`

Helper class for database objects with `OPTIONS` clauses

`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

**Parameters** `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.diff_map` (*inobj*)

Generate SQL to transform an existing object

**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 PostgreSQL foreign data wrapper. For PostgreSQL versions 9.1 and later see also [Foreign Data](#) and [Writing A Foreign Data Wrapper](#).

```
class pyseas.dbobject.foreign.ForeignDataWrapper (name=None, description=None,
                                                owner=None, privileges=None, **at-
                                                trs)
```

A foreign data wrapper definition

`ForeignDataWrapper.to_map` (*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` (*obj, \*args, \*\*kwargs*)

Return SQL statements to CREATE the data wrapper

**Returns** SQL statements

`ForeignDataWrapper.diff_map` (*inwrapper*)

Generate SQL to transform an existing wrapper

**Parameters** `inwrapper` – a YAML map defining the new wrapper

**Returns** list of 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 pyseas.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

`ForeignDataWrapperDict.link_refs` (*dbservers*)

Connect servers to their respective foreign data wrappers

**Parameters** **dbservers** – dictionary of foreign servers

`ForeignDataWrapperDict.diff_map` (*inwrappers*)

Generate SQL to transform existing data wrappers

**Parameters** **input\_map** – a YAML map defining the new data wrappers

**Returns** list of SQL statements

Compares the existing data wrapper definitions, as fetched from the catalogs, to the input map and generates SQL statements to transform the data wrappers accordingly.

## Foreign Server

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

**class** `pyseas.dboject.foreign.ForeignServer` (*name=None, description=None, owner=None, privileges=None, \*\*attrs*)

A foreign server definition

`ForeignServer.identifier` ()

Returns a full identifier for the foreign server

**Returns** string

`ForeignServer.to_map` (*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` (*obj, \*args, \*\*kwargs*)

Return SQL statements to CREATE the server

**Returns** SQL statements

`ForeignServer.diff_map` (*inserver*)

Generate SQL to transform an existing server

**Parameters** **inserver** – a YAML map defining the new server

**Returns** list of 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** `pyseas.dboject.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** (*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.

ForeignServerDict.**link\_refs** (*dbusermaps*)  
Connect user mappings to their respective servers

**Parameters** **dbusermaps** – dictionary of user mappings

ForeignServerDict.**diff\_map** (*inservers*)  
Generate SQL to transform existing foreign servers

**Parameters** **inservers** – a YAML map defining the new foreign servers

**Returns** list of SQL statements

Compares the existing server definitions, as fetched from the catalogs, to the input map and generates SQL statements to transform the foreign servers accordingly.

## User Mapping

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

**class** pyrseas.dbobject.foreign.**UserMapping** (*name=None, description=None, owner=None, privileges=None, \*\*attrs*)

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** ()  
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` ()

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.

`UserMappingDict.diff_map` (*inusermaps*)

Generate SQL to transform existing user mappings

**Parameters** **input\_map** – a YAML map defining the new user mappings

**Returns** list of SQL statements

Compares the existing user mapping definitions, as fetched from the catalogs, to the input map and generates SQL statements to transform the user mappings accordingly.

## Foreign Table

*ForeignTable* is derived from *DBObjectWithOptions* and *Table*. It represents a PostgreSQL foreign table (available on PostgreSQL 9.1 or later).

**class** `pyrseas.dbobject.foreign.ForeignTable` (*schema, name, description=None, owner=None, privileges=None, \*\*attrs*)

A foreign table definition

`ForeignTable.to_map` (*opts*)

Convert a foreign table to a YAML-suitable format

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

**Returns** dictionary

`ForeignTable.create` (*obj, \*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

`ForeignTable.diff_map` (*intable*)

Generate SQL to transform an existing table

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

**Returns** list of SQL statements

## Foreign Table Dictionary

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

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

`ForeignTableDict`.**from\_map** (*schema, inobjs, newdb*)  
 Initialize 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

`ForeignTableDict`.**link\_refs** (*dbcolumns*)  
 Connect columns to their respective foreign tables

**Parameters** **dbcolumns** – dictionary of columns

`ForeignTableDict`.**diff\_map** (*intables*)  
 Generate SQL to transform existing foreign tables

**Parameters** **intables** – a YAML map defining the new foreign tables

**Returns** list of SQL statements

Compares the existing foreign table definitions, as fetched from the catalogs, to the input map and generates SQL statements to transform the foreign tables accordingly.



## Augmenter Databases

The `augmentdb` module defines *AugmentDatabase*.

### 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.

## Augmenter Configuration Objects

These configuration objects are predefined in the Augmenter modules or can be defined or overridden by configuration elements in the augmenter map.

### Configuration Functions

A *CfgFunction* class specifies a 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 pyrsneas.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 pyrsneas.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 pyrsneas.augment.function.CfgFunctionSource (**attrs)
    A configuration function source or part thereof
```

```
class pyrsneas.augment.function.CfgFunctionTemplate (**attrs)
    A configuration function source template
```

```
class pyrsneas.augment.function.CfgFunctionSourceDict (cfg_templates)
```

### 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 pyrsneas.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** `pyrease.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

## 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** `pyrease.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** `pyrease.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

## 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** `pyrease.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** `pyrease.augment.audit.CfgAuditColumnDict` (*config*)  
 The collection of audit column augmentations

`CfgAuditColumnDict.from_map` (*inaudcols*)  
 Initialize the dictionary of functions by converting the input map

**Parameters** `inaudcols` – YAML map defining the audit column configuration

## 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.

## 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.

## 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*)

Initialize 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**

`pyrseas.augment.audit`, 75  
`pyrseas.augment.column`, 74  
`pyrseas.augment.function`, 74  
`pyrseas.augment.schema`, 75  
`pyrseas.augment.table`, 76  
`pyrseas.augment.trigger`, 75  
`pyrseas.augmentdb`, 73  
`pyrseas.database`, 35  
`pyrseas.dbobject`, 31  
`pyrseas.dbobject.cast`, 36  
`pyrseas.dbobject.collation`, 40  
`pyrseas.dbobject.column`, 55  
`pyrseas.dbobject.constraint`, 57  
`pyrseas.dbobject.conversion`, 41  
`pyrseas.dbobject.dbtype`, 48  
`pyrseas.dbobject.eventtrig`, 41  
`pyrseas.dbobject.extension`, 42  
`pyrseas.dbobject.foreign`, 66  
`pyrseas.dbobject.function`, 43  
`pyrseas.dbobject.index`, 60  
`pyrseas.dbobject.language`, 37  
`pyrseas.dbobject.operator`, 45  
`pyrseas.dbobject.operclass`, 47  
`pyrseas.dbobject.operfamily`, 46  
`pyrseas.dbobject.rule`, 61  
`pyrseas.dbobject.schema`, 38  
`pyrseas.dbobject.table`, 51  
`pyrseas.dbobject.textsearch`, 63  
`pyrseas.dbobject.trigger`, 62



## Symbols

- config <config-file>  
command line option, 28
  - exclude-schema <schema>  
dbtoyaml command line option, 25
  - exclude-table <table>  
dbtoyaml command line option, 25
  - host <host>  
command line option, 28
  - output <file>  
command line option, 28
  - port <port>  
command line option, 28
  - quote-reserved  
yamltodb command line option, 27
  - repository <path>  
command line option, 28
  - revert  
yamltodb command line option, 27
  - schema <schema>  
dbtoyaml command line option, 25  
yamltodb command line option, 27
  - single-transaction  
yamltodb command line option, 27
  - table <table>  
dbtoyaml command line option, 25
  - user <username>  
command line option, 28
  - version  
command line option, 28
  - l  
yamltodb command line option, 27
  - H <host>  
command line option, 28
  - N <schema>  
dbtoyaml command line option, 25
  - O, -no-owner  
dbtoyaml command line option, 25
  - T <table>  
dbtoyaml command line option, 25
  - U <username>  
command line option, 28
  - W, -password  
command line option, 28
  - c <config-file>  
command line option, 28
  - h, -help  
command line option, 28
  - m, -multiple-files  
dbtoyaml command line option, 25  
yamltodb command line option, 27
  - n <schema>  
dbtoyaml command line option, 25  
yamltodb command line option, 27
  - o <file>  
command line option, 28
  - p <port>  
command line option, 28
  - r <path>  
command line option, 28
  - t <table>  
dbtoyaml command line option, 25
  - u, -update  
yamltodb command line option, 27
  - x, -no-privileges  
dbtoyaml command line option, 25
- ## A
- add() (pyrseas.dboobject.column.Column method), 56
  - add() (pyrseas.dboobject.constraint.CheckConstraint method), 58
  - add() (pyrseas.dboobject.constraint.Constraint method), 57
  - add() (pyrseas.dboobject.constraint.ForeignKey method), 59
  - add\_owner() (pyrseas.dboobject.table.Sequence method), 52
  - add\_privs() (pyrseas.dboobject.column.Column method), 56
  - Aggregate (class in pyrseas.dboobject.function), 44

alter\_owner() (pyrseas.dbobject.DbObject method), 33  
 apply() (pyrseas.augment.audit.CfgAuditColumn method), 75  
 apply() (pyrseas.augment.column.CfgColumn method), 74  
 apply() (pyrseas.augment.function.CfgFunction method), 74  
 apply() (pyrseas.augment.schema.AugSchema method), 76  
 apply() (pyrseas.augment.table.AugTable method), 76  
 apply() (pyrseas.augment.trigger.CfgTrigger method), 75  
 apply() (pyrseas.augmentdb.AugmentDatabase method), 73

AugClassDict (class in pyrseas.augment.table), 76  
 AugDbClass (class in pyrseas.augment.table), 76  
 AugmentDatabase (class in pyrseas.augmentdb), 73  
 AugSchema (class in pyrseas.augment.schema), 76  
 AugSchemaDict (class in pyrseas.augment.schema), 76  
 AugTable (class in pyrseas.augment.table), 76

## B

BaseType (class in pyrseas.dbobject.dbtype), 48

## C

Cast (class in pyrseas.dbobject.cast), 36  
 CastDict (class in pyrseas.dbobject.cast), 37  
 CfgAuditColumn (class in pyrseas.augment.audit), 75  
 CfgAuditColumnDict (class in pyrseas.augment.audit), 75  
 CfgColumn (class in pyrseas.augment.column), 74  
 CfgColumnDict (class in pyrseas.augment.column), 74  
 CfgFunction (class in pyrseas.augment.function), 74  
 CfgFunctionDict (class in pyrseas.augment.function), 74  
 CfgFunctionSource (class in pyrseas.augment.function), 74  
 CfgFunctionSourceDict (class in pyrseas.augment.function), 74  
 CfgFunctionTemplate (class in pyrseas.augment.function), 74  
 CfgTrigger (class in pyrseas.augment.trigger), 75  
 CfgTriggerDict (class in pyrseas.augment.trigger), 75  
 CheckConstraint (class in pyrseas.dbobject.constraint), 58  
 ClassDict (class in pyrseas.dbobject.table), 55  
 cls (pyrseas.dbobject.DbObjectDict attribute), 34  
 Collation (class in pyrseas.dbobject.collation), 40  
 CollationDict (class in pyrseas.dbobject.collation), 40  
 Column (class in pyrseas.dbobject.column), 56  
 column\_names() (pyrseas.dbobject.table.Table method), 53  
 ColumnDict (class in pyrseas.dbobject.column), 57  
 command line option  
     –config <config-file>, 28  
     –host <host>, 28

    –output <file>, 28  
     –port <port>, 28  
     –repository <path>, 28  
     –user <username>, 28  
     –version, 28  
     –H <host>, 28  
     –U <username>, 28  
     –W, –password, 28  
     –c <config-file>, 28  
     –h, –help, 28  
     –o <file>, 28  
     –p <port>, 28  
     –r <path>, 28  
 comment() (pyrseas.dbobject.column.Column method), 56  
 comment() (pyrseas.dbobject.constraint.Constraint method), 58  
 comment() (pyrseas.dbobject.DbObject method), 33  
 Composite (class in pyrseas.dbobject.dbtype), 49  
 Constraint (class in pyrseas.dbobject.constraint), 57  
 ConstraintDict (class in pyrseas.dbobject.constraint), 60  
 Conversion (class in pyrseas.dbobject.conversion), 41  
 ConversionDict (class in pyrseas.dbobject.conversion), 41  
 create() (pyrseas.dbobject.cast.Cast method), 36  
 create() (pyrseas.dbobject.collation.Collation method), 40  
 create() (pyrseas.dbobject.conversion.Conversion method), 41  
 create() (pyrseas.dbobject.dbtype.BaseType method), 49  
 create() (pyrseas.dbobject.dbtype.Composite method), 49  
 create() (pyrseas.dbobject.dbtype.Domain method), 50  
 create() (pyrseas.dbobject.dbtype.Enum method), 49  
 create() (pyrseas.dbobject.eventtrig.EventTrigger method), 42  
 create() (pyrseas.dbobject.extension.Extension method), 42  
 create() (pyrseas.dbobject.foreign.ForeignDataWrapper method), 67  
 create() (pyrseas.dbobject.foreign.ForeignServer method), 68  
 create() (pyrseas.dbobject.foreign.ForeignTable method), 70  
 create() (pyrseas.dbobject.foreign.UserMapping method), 69  
 create() (pyrseas.dbobject.function.Aggregate method), 44  
 create() (pyrseas.dbobject.function.Function method), 44  
 create() (pyrseas.dbobject.index.Index method), 61  
 create() (pyrseas.dbobject.language.Language method), 37  
 create() (pyrseas.dbobject.operator.Operator method), 45  
 create() (pyrseas.dbobject.operclass.OperatorClass method), 47  
 create() (pyrseas.dbobject.operfamily.OperatorFamily method), 46



- create() (pyrseas.dboject.rule.Rule method), 62
  - create() (pyrseas.dboject.schema.Schema method), 39
  - create() (pyrseas.dboject.table.Sequence method), 51
  - create() (pyrseas.dboject.table.Table method), 53
  - create() (pyrseas.dboject.table.View method), 54
  - create() (pyrseas.dboject.textsearch.TSConfiguration method), 64
  - create() (pyrseas.dboject.textsearch.TSDictionary method), 64
  - create() (pyrseas.dboject.textsearch.TSParser method), 65
  - create() (pyrseas.dboject.textsearch.TSTemplate method), 65
  - create() (pyrseas.dboject.trigger.Trigger method), 63
- D**
- data\_export() (pyrseas.dboject.table.Table method), 53
  - data\_import() (pyrseas.dboject.schema.Schema method), 39
  - data\_import() (pyrseas.dboject.schema.SchemaDict method), 40
  - data\_import() (pyrseas.dboject.table.Table method), 53
  - Database (class in pyrseas.database), 35
  - DbClass (class in pyrseas.dboject.table), 51
  - DBObject (class in pyrseas.dboject), 31
  - DBObjectDict (class in pyrseas.dboject), 33
  - DBObjectWithOptions (class in pyrseas.dboject.foreign), 66
  - DbSchemaObject (class in pyrseas.dboject), 34
  - dbtoyaml command line option
    - exclude-schema <schema>, 25
    - exclude-table <table>, 25
    - schema <schema>, 25
    - table <table>, 25
    - N <schema>, 25
    - O, -no-owner, 25
    - T <table>, 25
    - m, -multiple-files, 25
    - n <schema>, 25
    - t <table>, 25
    - x, -no-privileges, 25
  - DbType (class in pyrseas.dboject.dbtype), 48
  - diff\_description() (pyrseas.dboject.DbObject method), 33
  - diff\_map() (pyrseas.database.Database method), 36
  - diff\_map() (pyrseas.dboject.cast.Cast method), 36
  - diff\_map() (pyrseas.dboject.cast.CastDict method), 37
  - diff\_map() (pyrseas.dboject.collation.CollationDict method), 40
  - diff\_map() (pyrseas.dboject.column.Column method), 56
  - diff\_map() (pyrseas.dboject.column.ColumnDict method), 57
  - diff\_map() (pyrseas.dboject.constraint.CheckConstraint method), 58
  - diff\_map() (pyrseas.dboject.constraint.ConstraintDict method), 60
  - diff\_map() (pyrseas.dboject.constraint.ForeignKey method), 59
  - diff\_map() (pyrseas.dboject.constraint.PrimaryKey method), 58
  - diff\_map() (pyrseas.dboject.constraint.UniqueConstraint method), 60
  - diff\_map() (pyrseas.dboject.conversion.ConversionDict method), 41
  - diff\_map() (pyrseas.dboject.DbObject method), 33
  - diff\_map() (pyrseas.dboject.dbtype.Composite method), 49
  - diff\_map() (pyrseas.dboject.dbtype.TypeDict method), 50
  - diff\_map() (pyrseas.dboject.eventtrig.EventTriggerDict method), 42
  - diff\_map() (pyrseas.dboject.extension.ExtensionDict method), 43
  - diff\_map() (pyrseas.dboject.foreign.DbObjectWithOptions method), 67
  - diff\_map() (pyrseas.dboject.foreign.ForeignDataWrapper method), 67
  - diff\_map() (pyrseas.dboject.foreign.ForeignDataWrapperDict method), 68
  - diff\_map() (pyrseas.dboject.foreign.ForeignServer method), 68
  - diff\_map() (pyrseas.dboject.foreign.ForeignServerDict method), 69
  - diff\_map() (pyrseas.dboject.foreign.ForeignTable method), 70
  - diff\_map() (pyrseas.dboject.foreign.ForeignTableDict method), 71
  - diff\_map() (pyrseas.dboject.foreign.UserMappingDict method), 70
  - diff\_map() (pyrseas.dboject.function.Function method), 44
  - diff\_map() (pyrseas.dboject.function.ProcDict method), 45
  - diff\_map() (pyrseas.dboject.index.Index method), 61
  - diff\_map() (pyrseas.dboject.index.IndexDict method), 61
  - diff\_map() (pyrseas.dboject.language.LanguageDict method), 38
  - diff\_map() (pyrseas.dboject.operator.OperatorDict method), 46
  - diff\_map() (pyrseas.dboject.operclass.OperatorClassDict method), 48
  - diff\_map() (pyrseas.dboject.operfamily.OperatorFamilyDict method), 47
  - diff\_map() (pyrseas.dboject.rule.RuleDict method), 62
  - diff\_map() (pyrseas.dboject.schema.SchemaDict method), 39

method), 39  
diff\_map() (pyrseas.dboject.table.ClassDict method), 55  
diff\_map() (pyrseas.dboject.table.MaterializedView method), 54  
diff\_map() (pyrseas.dboject.table.Sequence method), 52  
diff\_map() (pyrseas.dboject.table.Table method), 53  
diff\_map() (pyrseas.dboject.table.View method), 54  
diff\_map() (pyrseas.dboject.textsearch.TSConfigurationDict method), 64  
diff\_map() (pyrseas.dboject.textsearch.TSDictionaryDict method), 65  
diff\_map() (pyrseas.dboject.textsearch.TSParserDict method), 65  
diff\_map() (pyrseas.dboject.textsearch.TSTemplateDict method), 66  
diff\_map() (pyrseas.dboject.trigger.TriggerDict method), 63  
diff\_options() (pyrseas.dboject.foreign.DbObjectWithOptions method), 66  
diff\_options() (pyrseas.dboject.table.Table method), 53  
diff\_privileges() (pyrseas.dboject.column.Column method), 56  
diff\_privileges() (pyrseas.dboject.DbObject method), 33  
Domain (class in pyrseas.dboject.dbtype), 50  
drop() (pyrseas.dboject.column.Column method), 56  
drop() (pyrseas.dboject.constraint.Constraint method), 57  
drop() (pyrseas.dboject.DbObject method), 33  
drop() (pyrseas.dboject.DbSchemaObject method), 34  
drop() (pyrseas.dboject.dbtype.BaseType method), 49  
drop() (pyrseas.dboject.foreign.ForeignTable method), 70  
drop() (pyrseas.dboject.table.Table method), 53

## E

Enum (class in pyrseas.dboject.dbtype), 49  
EventTrigger (class in pyrseas.dboject.eventtrig), 41  
EventTriggerDict (class in pyrseas.dboject.eventtrig), 42  
Extension (class in pyrseas.dboject.extension), 42  
ExtensionDict (class in pyrseas.dboject.extension), 43  
extern\_dir() (pyrseas.dboject.schema.Schema method), 38  
extern\_filename() (pyrseas.dboject.DbObject method), 32  
extern\_key() (pyrseas.dboject.cast.Cast method), 36  
extern\_key() (pyrseas.dboject.DbObject method), 32  
extern\_key() (pyrseas.dboject.foreign.UserMapping method), 69  
extern\_key() (pyrseas.dboject.function.Proc method), 43  
extern\_key() (pyrseas.dboject.operator.Operator method), 45  
extern\_key() (pyrseas.dboject.operclass.OperatorClass method), 47

extern\_key() (pyrseas.dboject.operfamily.OperatorFamily method), 46

## F

fetch() (pyrseas.dboject.DbObjectDict method), 34  
ForeignDataWrapper (class in pyrseas.dboject.foreign), 67  
ForeignDataWrapperDict (class in pyrseas.dboject.foreign), 67  
ForeignKey (class in pyrseas.dboject.constraint), 59  
ForeignServer (class in pyrseas.dboject.foreign), 68  
ForeignServerDict (class in pyrseas.dboject.foreign), 68  
ForeignTable (class in pyrseas.dboject.foreign), 70  
ForeignTableDict (class in pyrseas.dboject.foreign), 71  
from\_augmap() (pyrseas.augmentdb.AugmentDatabase method), 73  
from\_catalog() (pyrseas.database.Database method), 35  
from\_map() (pyrseas.augment.audit.CfgAuditColumnDict method), 75  
from\_map() (pyrseas.augment.column.CfgColumnDict method), 75  
from\_map() (pyrseas.augment.function.CfgFunctionDict method), 74  
from\_map() (pyrseas.augment.schema.AugSchemaDict method), 76  
from\_map() (pyrseas.augment.table.AugClassDict method), 76  
from\_map() (pyrseas.augment.trigger.CfgTriggerDict method), 75  
from\_map() (pyrseas.database.Database method), 35  
from\_map() (pyrseas.dboject.cast.CastDict method), 37  
from\_map() (pyrseas.dboject.collation.CollationDict method), 40  
from\_map() (pyrseas.dboject.column.ColumnDict method), 57  
from\_map() (pyrseas.dboject.constraint.ConstraintDict method), 60  
from\_map() (pyrseas.dboject.conversion.ConversionDict method), 41  
from\_map() (pyrseas.dboject.dbtype.TypeDict method), 50  
from\_map() (pyrseas.dboject.eventtrig.EventTriggerDict method), 42  
from\_map() (pyrseas.dboject.extension.ExtensionDict method), 43  
from\_map() (pyrseas.dboject.foreign.ForeignDataWrapperDict method), 67  
from\_map() (pyrseas.dboject.foreign.ForeignServerDict method), 68  
from\_map() (pyrseas.dboject.foreign.ForeignTableDict method), 71  
from\_map() (pyrseas.dboject.foreign.UserMappingDict method), 70



ProcDict (class in `pyrseas.dboject.function`), 44  
`pyrseas.augment.audit` (module), 75  
`pyrseas.augment.column` (module), 74  
`pyrseas.augment.function` (module), 74  
`pyrseas.augment.schema` (module), 75  
`pyrseas.augment.table` (module), 76  
`pyrseas.augment.trigger` (module), 75  
`pyrseas.augmentdb` (module), 73  
`pyrseas.database` (module), 35  
`pyrseas.dboject` (module), 31  
`pyrseas.dboject.cast` (module), 36  
`pyrseas.dboject.collation` (module), 40  
`pyrseas.dboject.column` (module), 55  
`pyrseas.dboject.constraint` (module), 57  
`pyrseas.dboject.conversion` (module), 41  
`pyrseas.dboject.dbtype` (module), 48  
`pyrseas.dboject.eventtrig` (module), 41  
`pyrseas.dboject.extension` (module), 42  
`pyrseas.dboject.foreign` (module), 66  
`pyrseas.dboject.function` (module), 43  
`pyrseas.dboject.index` (module), 60  
`pyrseas.dboject.language` (module), 37  
`pyrseas.dboject.operator` (module), 45  
`pyrseas.dboject.operclass` (module), 47  
`pyrseas.dboject.operfamily` (module), 46  
`pyrseas.dboject.rule` (module), 61  
`pyrseas.dboject.schema` (module), 38  
`pyrseas.dboject.table` (module), 51  
`pyrseas.dboject.textsearch` (module), 63  
`pyrseas.dboject.trigger` (module), 62

## Q

`qualname()` (`pyrseas.dboject.DbSchemaObject` method), 34  
`qualname()` (`pyrseas.dboject.operator.Operator` method), 45  
`query` (`pyrseas.dboject.DbObjectDict` attribute), 34

## R

`ref_columns()` (`pyrseas.dboject.constraint.ForeignKey` method), 59  
`rename()` (`pyrseas.dboject.column.Column` method), 56  
`rename()` (`pyrseas.dboject.DbObject` method), 33  
`rename()` (`pyrseas.dboject.DbSchemaObject` method), 34  
Rule (class in `pyrseas.dboject.rule`), 62  
RuleDict (class in `pyrseas.dboject.rule`), 62

## S

Schema (class in `pyrseas.dboject.schema`), 38  
SchemaDict (class in `pyrseas.dboject.schema`), 39  
Sequence (class in `pyrseas.dboject.table`), 51  
`set_sequence_default()` (`pyrseas.dboject.column.Column` method), 56

## T

Table (class in `pyrseas.dboject.table`), 52  
`to_map()` (`pyrseas.database.Database` method), 36  
`to_map()` (`pyrseas.dboject.cast.Cast` method), 36  
`to_map()` (`pyrseas.dboject.column.Column` method), 56  
`to_map()` (`pyrseas.dboject.constraint.CheckConstraint` method), 58  
`to_map()` (`pyrseas.dboject.constraint.ForeignKey` method), 59  
`to_map()` (`pyrseas.dboject.constraint.PrimaryKey` method), 58  
`to_map()` (`pyrseas.dboject.constraint.UniqueConstraint` method), 59  
`to_map()` (`pyrseas.dboject.DbObject` method), 32  
`to_map()` (`pyrseas.dboject.DbObjectDict` method), 34  
`to_map()` (`pyrseas.dboject.dbtype.BaseType` method), 48  
`to_map()` (`pyrseas.dboject.dbtype.Composite` method), 49  
`to_map()` (`pyrseas.dboject.dbtype.Domain` method), 50  
`to_map()` (`pyrseas.dboject.foreign.ForeignDataWrapper` method), 67  
`to_map()` (`pyrseas.dboject.foreign.ForeignServer` method), 68  
`to_map()` (`pyrseas.dboject.foreign.ForeignServerDict` method), 69  
`to_map()` (`pyrseas.dboject.foreign.ForeignTable` method), 70  
`to_map()` (`pyrseas.dboject.foreign.UserMappingDict` method), 70  
`to_map()` (`pyrseas.dboject.function.Aggregate` method), 44  
`to_map()` (`pyrseas.dboject.function.Function` method), 43  
`to_map()` (`pyrseas.dboject.index.Index` method), 61  
`to_map()` (`pyrseas.dboject.language.Language` method), 37  
`to_map()` (`pyrseas.dboject.operclass.OperatorClass` method), 47  
`to_map()` (`pyrseas.dboject.rule.Rule` method), 62  
`to_map()` (`pyrseas.dboject.schema.Schema` method), 38  
`to_map()` (`pyrseas.dboject.schema.SchemaDict` method), 39  
`to_map()` (`pyrseas.dboject.table.MaterializedView` method), 54  
`to_map()` (`pyrseas.dboject.table.Sequence` method), 51  
`to_map()` (`pyrseas.dboject.table.Table` method), 53  
`to_map()` (`pyrseas.dboject.table.View` method), 54  
`to_map()` (`pyrseas.dboject.textsearch.TSConfiguration` method), 63  
`to_map()` (`pyrseas.dboject.trigger.Trigger` method), 63  
Trigger (class in `pyrseas.dboject.trigger`), 62  
TriggerDict (class in `pyrseas.dboject.trigger`), 63

TSConfiguration (class in pyrseas.dbobject.textsearch),  
63

TSConfigurationDict (class in  
pyrseas.dbobject.textsearch), 64

TSDictionary (class in pyrseas.dbobject.textsearch), 64

TSDictionaryDict (class in pyrseas.dbobject.textsearch),  
64

TSParser (class in pyrseas.dbobject.textsearch), 65

TSParserDict (class in pyrseas.dbobject.textsearch), 65

TSTemplate (class in pyrseas.dbobject.textsearch), 65

TSTemplateDict (class in pyrseas.dbobject.textsearch), 66

TypeDict (class in pyrseas.dbobject.dbtype), 50

## U

UniqueConstraint (class in pyrseas.dbobject.constraint),  
59

unqualify() (pyrseas.dbobject.DbSchemaObject method),  
34

UserMapping (class in pyrseas.dbobject.foreign), 69

UserMappingDict (class in pyrseas.dbobject.foreign), 70

## V

View (class in pyrseas.dbobject.table), 54

## Y

yamltodb command line option

  –quote-reserved, 27

  –revert, 27

  –schema <schema>, 27

  –single-transaction, 27

  -1, 27

  -m, –multiple-files, 27

  -n <schema>, 27

  -u, –update, 27