
PUDL

Release 0.2.0

Catalyst Cooperative

Sep 18, 2019

CONTENTS

1	Getting Started	3
2	Contributing to PUDL	5
3	Licensing	7
4	Contact Us	9
5	About Catalyst Cooperative	11
5.1	Installation and Setup	11
5.2	Basic Usage	14
5.3	Creating a Datastore	15
5.4	Cloning the FERC Form 1 DB	15
5.5	Published Data Packages	16
5.6	Cloud Based Access	18
5.7	Data Catalog	19
5.8	Settings Files	23
5.9	Contributing to PUDL	25
5.10	Development Setup	26
5.11	Building and Testing PUDL	30
5.12	Data and ETL Design Guidelines	35
5.13	Integrating a New Dataset	38
5.14	Code Standards	40
5.15	Project Management	41
5.16	Naming Conventions	42
5.17	Project Background	44
5.18	Acknowledgments	45
5.19	The MIT License	46
5.20	Catalyst Cooperative Code of Conduct	46
5.21	pudl	47
	Python Module Index	133
	Index	135

PUDL makes US energy data easier to access and work with. Hundreds of gigabytes of public information is published by government agencies, but in many different formats that make it hard to work with and combine. PUDL takes these spreadsheets, CSV files, and databases and turns them into easy use [tabular data packages](#) that can populate a database, or be used directly with Python, R, Microsoft Access, and many other tools.

The project currently integrates data from:

- [EIA Form 860](#)
- [EIA Form 923](#)
- [The EPA Continuous Emissions Monitoring System \(CEMS\)](#)
- [The EPA Integrated Planning Model \(IPM\)](#)
- [FERC Form 1](#)

The project is especially meant to serve researchers, activists, journalists, and policy makers that might not otherwise be able to afford access to this data from existing commercial data providers.

GETTING STARTED

Just want to play with some example data? Install [Anaconda](#) (or [miniconda](#)) with at least Python 3.7. Then work through the following commands.

First, we create and activate conda environment named `pudl`. All the required packages are available from the community maintained `conda-forge` channel, and that channel is given priority, to simplify satisfying dependencies. Note that PUDL currently requires Python 3.7, the most recently available major version of Python. In addition to the `catalystcoop.pudl` package, we'll also install JupyterLab so we can work with the PUDL data interactively.

```
$ conda create -y -n pudl -c conda-forge --strict-channel-priority python=3.7_
→catalystcoop.pudl jupyter jupyterlab pip
$ conda activate pudl
```

Now we create a data management workspace called `pudl-work` and download some data. The workspace is a well defined directory structure that PUDL uses to organize the data it downloads, processes, and outputs. You can run `pudl_setup --help` and `pudl_data --help` for more information.

```
$ mkdir pudl-work
$ pudl_setup pudl-work
$ pudl_data --sources eia923 eia860 fercl epacems epaipm --years 2017 --states id
```

Now that we have the original data as published by the federal agencies, we can run the ETL (Extract, Transform, Load) pipeline, that turns the raw data into an well organized, standardized bundle of data packages. This involves a couple of steps: cloning the FERC Form 1 into an SQLite database, extracting data from that database and all the other sources and cleaning it up, outputting that data into well organized CSV/JSON based data packages, and finally loading those data packages into a local database.

PUDL provides a script to clone the FERC Form 1 database, controlled by a YAML file which you can find in the settings folder. Run it like this:

```
$ fercl_to_sqlite pudl-work/settings/fercl_to_sqlite_example.yml
```

The main ETL process is controlled by the YAML file `etl_example.yml` which defines what data will be processed. It is well commented – if you want to understand what the options are, open it in a text editor, and create your own version.

The data packages will be generated in a sub-directory in `pudl-work/datapackage` named `pudl-example` (you can change this by changing the value of `pkg_bundle_name` in the ETL settings file you're using. Run the ETL pipeline with this command:

```
$ pudl_etl pudl-work/settings/etl_example.yml
```

The generated data packages are made up of CSV and JSON files. They're both easy to parse programmatically, and readable by humans. They are also well suited to archiving, citation, and bulk distribution. However, to make the data

easier to query and work with interactively, we typically load it into a local SQLite database using this script, which combines several data packages from the same bundle into a single unified structure:

```
$ datapkg_to_sqlite --pkg_bundle_name pudl-example
```

Now that we have a live database, we can easily work with it using a variety of tools, including Python, pandas dataframes, and [Jupyter notebooks](#). This command will start up a local Jupyter notebook server, and open a notebook of PUDL usage examples:

```
$ jupyter lab pudl-work/notebook/pudl_intro.ipynb
```

For more details, see [the full PUDL documentation](#) on Read The Docs.

CONTRIBUTING TO PUDL

Find PUDL useful? Want to help make it better? There are lots of ways to contribute!

- Please be sure to read our [Code of Conduct](#)
- You can file a bug report, make a feature request, or ask questions in the [Github issue tracker](#).
- Feel free to fork the project and make a pull request with new code, better documentation, or example notebooks.
- [Make a recurring financial contribution](#) to support our work liberating public energy data.
- Hire us to do some custom analysis, and let us add the code the project.
- For more information check out our [Contribution Guidelines](#)

LICENSING

The PUDL software is released under the [MIT License](#). The PUDL documentation and the data packages we distribute are released under the [CC-BY-4.0](#) license.

CONTACT US

For help with initial setup, usage questions, bug reports, suggestions to make PUDL better and anything else that could conceivably be of use or interest to the broader community of users, use the [PUDL issue tracker](#). on Github. For private communication about the project, you can email the team: pudl@catalyst.coop

ABOUT CATALYST COOPERATIVE

Catalyst Cooperative is a small group of data scientists and policy wonks. We're organized as a worker-owned cooperative consultancy. Our goal is a more just, livable, and sustainable world. We integrate public data and perform custom analyses to inform public policy making. Our focus is primarily on mitigating climate change and improving electric utility regulation in the United States.

Do you work on renewable energy or climate policy? Have you found yourself scraping data from government PDFs, spreadsheets, websites, and databases, without getting something reusable? We build tools to pull this kind of information together reliably and automatically so you can focus on your real work instead — whether that's political advocacy, energy journalism, academic research, or public policy making.

- Web: <https://catalyst.coop>
- Newsletter: <https://catalyst.coop/updates/>
- Email: hello@catalyst.coop
- Twitter: [@CatalystCoop](https://twitter.com/CatalystCoop)

5.1 Installation and Setup

5.1.1 System Requirements

Note: The PUDL data processing pipeline does a lot of work in-memory with `pandas.DataFrame` objects. The full *EPA CEMS Hourly* dataset is nearly 100 GB uncompressed. To handle all of the data that is available via PUDL we recommend that your system have at least:

- **8 GB of memory**
 - **100 GB of free disk space**
-

Python 3.7+ (and conda)

PUDL requires Python 3.7 or later. In addition, while not strictly necessary, we **highly** recommend using the most recent version of the [Anaconda Python distribution](#), or its smaller cousin [miniconda](#) (miniconda is nice if you are fond of the command line and want a lightweight install).

Both Anaconda and miniconda provide `conda`, a command-line tool that helps you manage your Python software environment, packages, and their dependencies. PUDL provides an `environment.yml` file defining a software environment that should work well for most users in conjunction with `conda`.

We recommend using `conda` because while PUDL is written entirely in Python, it makes heavy use of Python's open data science stack including packages like `numpy`, `scipy`, `pandas`, and `sklearn` which depend on extensions written in C and C++. These extensions can be difficult to build locally when installed with `pip`, but `conda` provides pre-compiled platform specific binaries.

5.1.2 Installing the Package

PUDL and all of its dependencies are available via `conda` on the community managed `conda-forge` channel, and we recommend installing PUDL within its own `conda` environment like this:

```
$ conda create -y -n pudl -c conda-forge --strict-channel-priority python=3.7.
↳catalystcoop.pudl pip
```

Then you activate that `conda` environment to use it:

```
$ conda activate pudl
```

Once you've activated that environment, you may want to install additional software within it, for example if you want to use Jupyter notebooks to work with PUDL interactively:

```
$ conda install jupyter jupyterlab
```

You may also want to update your global `conda` settings:

```
$ conda config --add channels conda-forge
$ conda config --set channel_priority strict
```

PUDL is also available via the official [Python Package Index \(PyPI\)](#) and be installed with `pip` like this:

```
$ pip install catalystcoop.pudl
```

Note: `pip` will only install the dependencies required for PUDL to work as a development library and command line tool. If you want to check out the source code from Github for development purposes, see the [Development Setup](#) documentation.

In addition to making the `pudl` package available for import in Python, installing `catalystcoop.pudl` provides the following command line tools:

- `pudl_setup`
- `pudl_data`
- `fercl_to_sqlite`
- `pudl_etl`
- `datapkg_to_sqlite`
- `epacems_to_parquet`

For information on how to use these scripts, each can be run with the `--help` option. `fercl_to_sqlite` and `pudl_etl` are configured with YAML files. Examples are provided with the `catalystcoop.pudl` package, and deployed by running `pudl_setup` as described below. Additional information about the settings files can be found in our documentation on [Settings Files](#)

5.1.3 Creating a Workspace

PUDL needs to know where to store its big piles of inputs and outputs. It also provides some example configuration files and [Jupyter](#) notebooks. The `pudl_setup` script lets PUDL know where all this stuff should go. We call this a “PUDL workspace”:

```
$ pudl_setup <PUDL_DIR>
```

Here `<PUDL_DIR>` is the path to the directory where you want PUDL to do its business – this is where the datastore will be located, and any outputs that are generated will end up. The script will also put a configuration file in your home directory, called `.pudl.yml` that records the location of this workspace and uses it by default in the future. If you run `pudl_setup` with no arguments, it assumes you want to use the current directory.

The workspace is laid out like this:

Directory / File	Contents
<code>data/</code>	Raw data, automatically organized by source, year, etc.
<code>datapackage/</code>	Tabular data packages generated by PUDL.
<code>environment.yml</code>	A file describing the PUDL conda environment .
<code>notebook/</code>	Interactive Jupyter notebooks that use PUDL.
<code>parquet/</code>	Apache Parquet files generated by PUDL.
<code>settings/</code>	Example configuration files for controlling PUDL scripts.
<code>sqlite/</code>	sqlite3 databases generated by PUDL.

5.1.4 The PUDL conda Environment

In addition to creating a `conda` environment using the command line arguments referred to above you can specify an environment in a file, usually named `environment.yml`. We deploy a basic version of this file into a PUDL workspace when it’s created, as listed above.

Create the Environment

Because you won’t have the `environment.yml` file until after you’ve installed PUDL, you will probably create your PUDL environment on the command line as described above. To do the same thing using an environment file, you’d run:

```
$ conda env create --name pudl --file environment.yml
```

You may want to periodically update PUDL and the packages it depends on by running the following commands in the directory with `environment.yml` in it:

```
$ conda update conda
$ conda env update pudl
```

If you get an error `No such file or directory: environment.yml`, it probably means you aren’t in the same directory as the `environment.yml` file.

Activate the Environment

`conda` allows you to set up different software environments for different projects. However, this means you need to tell `conda` which environment you want to be using at any given time. To select a particular `conda` environment (like the one named `pudl` that you just created) use `conda activate` followed by the name of the environment you want to use:

```
$ conda activate pudl
```

After running this command you should see an indicator (like `(pudl)`) in your command prompt, signaling that the environment is in use.

See also:

[Managing Environments](#), in the `conda` documentation.

5.2 Basic Usage

PUDL implements a data processing pipeline. This pipeline takes raw data provided by public agencies in a variety of formats and integrates it together into a single (more) coherent whole. In the data-science world this is often called “ETL” which stands for “Extract, Transform, Load.”

- **Extract** the data from its original source formats and into `pandas.DataFrame` objects for easy manipulation.
- **Transform** the extracted data into tidy tabular data structures, applying a variety of cleaning routines, and creating connections both within and between the various datasets.
- **Load** the data into a standardized output, in our case CSV/JSON based [Tabular Data Packages](#), and potentially an SQLite database.

The PUDL python package is organized into these steps as well, with `pudl.extract` and `pudl.transform` sub-packages that contain dataset specific modules like `pudl.extract.ferc1` and `pudl.transform.eia923`. The Load step is handled by the `pudl.load`, subpackage, which contains modules that deal separately with generating CSVs containing the data, and JSON files containing the metadata.

We have also begun building a data validation step with the `pudl.validate` module, to catch any inadvertent data corruption and as-of-yet unfixed reporting errors.

The ETL pipeline is coordinated by the top-level `pudl.etl` module, which has a command line interface accessible via the `pudl_etl` script that is installed by the PUDL Python package. The script reads a YAML file as input. An example is provided in the `settings` folder that is created when you run `pudl_setup` (see: [Creating a Workspace](#)).

To run the ETL pipeline for the example, from within your PUDL workspace you would need to run four commands, which [download the original data](#), then [clone the FERC Form 1 database](#), convert that and other raw data into data-packages, and loads those datapackages into an SQLite database, respectively:

```
$ pudl_data --sources eia923 eia860 ferc1 epacems epaipm --years 2017 --states id
$ ferc1_to_sqlite settings/ferc1_to_sqlite_example.yml
$ pudl_etl settings/etl_example.yml
$ datapkg_to_sqlite --pkg_bundle_name pudl-example
```

These commands should result in a bunch of Python `logging` output, describing what the script is doing, and some outputs in the `sqlite` and `datapackage` directories within your workspace. In particular, you should see new files at `sqlite/ferc1.sqlite` and `sqlite/pudl.sqlite`, and a new directory at `datapackage/pudl-example` containing several `datapackage` directories, one for each of the `ferc1`, `eia` (Forms 860 and 923), `epacems-eia`, and `epaipm` datasets.

Under the hood, these scripts are extracting data from the datastore, including spreadsheets, CSV files, and binary DBF files, generating a SQLite database containing the raw FERC Form 1 data, and combining it all into `pudl-example`, which is a bundle of [tabular datapackages](#). that can be used together to create a database (or other things).

Each of the data packages which are part of the bundle have metadata describing their structure, stored in a file called `datapackage.json`. The data itself is stored in a bunch of CSV files (some of which may be `gzip` compressed) in the `data/` directories of each data package.

You can use the `pudl_etl` script to process more or different data by copying and editing the `settings/etl_example.yml` file, and running the script again with your new settings file as an argument. Comments in the example settings file explain the available parameters.

If you want to re-run `pudl_etl` and replace an existing bundle of data packages, you can use `--clobber`. If you want to generate a new data packages with a new or modified settings file, you can change the name for `--pkg_bundle_name` which will generate a new `datapackage/{your new name}` directory and will store your data packages there.

5.3 Creating a Datastore

The input data that PUDL processes comes from a variety of US government agencies. These agencies typically make the data available on their websites or via FTP without really planning for programmatic access.

The `pudl_data` script helps you obtain and organize this data locally, for use by the rest of the PUDL system. It uses the routines defined in the `pudl.workspace.datastore` module. For details on what data is available, for what time periods, and how much of it there is, see the [Data Catalog](#).

For example, if you wanted to download the 2018 *EPA CEMS Hourly* data for Colorado:

```
$ pudl_data --sources epacems --states CO --years 2018
```

If you do not specify years, the script will retrieve all available data. So to get everything for *EIA Form 860* and *EIA Form 923* you would run:

```
$ pudl_data --sources eia860 eia923
```

The script will download from all sources in parallel, so if you have a fast internet connection and need a lot of data, doing it all in one go makes sense. To pull down **all** the available data for all the sources (10+ GB) you would run:

```
$ pudl_data --sources eia860 eia923 epacems ferc1 epaipm
```

For more detailed usage information, see:

```
$ pudl_data --help
```

The downloaded data will be used by the script to populate a datastore under the `data` directory in your workspace, organized by data source, form, and date:

```
data/eia/form860/
data/eia/form923/
data/epa/cems/
data/epa/ipm/
data/ferc/form1/
```

If the download fails (e.g. the FTP server times out), this command can be run repeatedly until all the files are downloaded. It will not try and re-download data which is already present locally, unless you use the `--clobber` option. Depending on which data sources, how many years or states you have requested data for, and the speed of your internet connection, this may take minutes to hours to complete, and can consume 20+ GB of disk space even when the data is compressed.

5.4 Cloning the FERC Form 1 DB

FERC Form 1 is... special.

The *Form 1 data* is published in a particularly inaccessible format (proprietary binary FoxPro database files), and the data itself is unclean and poorly organized. As a result, very few people are currently able to use it at all, and we have not yet integrated the vast majority of the available data into PUDL. This also means it's useful to just provide programmatic access to the bulk raw data, independent of the cleaner subset of the data included within PUDL.

To provide that access, we've broken the `pudl.extract.ferc1`` process down into two distinct steps:

1. Clone the *entire* FERC Form 1 database from FoxPro into a local file-based `sqlite3` database. This includes 116 distinct tables, with thousands of fields, covering the time period from 1994 to the present.
2. Pull a subset of the data out of that database for further processing and integration into the PUDL data packages and `sqlite3` database.

If you want direct access to the original FERC Form 1 database, you can just do the database cloning, and connect directly to the resulting database. This has become especially useful since Microsoft recently discontinued the database driver that until late 2018 had allowed users to load the FoxPro database files into Microsoft Access.

In any case, cloning the original FERC database is the first step in the PUDL ETL process. This can be done with the `ferc1_to_sqlite` script (which is an entrypoint into the `pudl.convert.ferc1_to_sqlite` module) which is installed as part of the PUDL Python package. It takes its instructions from a YAML file, an example of which is included in the `settings` directory in your PUDL workspace. Once you've *created a datastore* you can try this example:

```
$ ferc1_to_sqlite settings/ferc1_to_sqlite_example.yml
```

This should create an SQLite database that you can find in your workspace at `sqlite/ferc1.sqlite`. By default, the script pulls in all available years of data, and all but 3 of the 100+ database tables. The excluded tables (`f1_footnote_tbl`, `f1_footnote_data` and `f1_note_fin_stmtnt`) contain unreadable binary data, and increase the overall size of the database by a factor of ~10 (to ~8 GB rather than 800 MB). If for some reason you need access to those tables, you can create your own settings file and un-comment those tables in the list of tables that it directs the script to load.

Note: This script pulls *all* of the FERC Form 1 data into a *single* database, but FERC distributes a *separate* database for each year. Virtually all the database tables contain a `report_year` column that indicates which year they came from, preventing collisions between records in the merged multi-year database. One notable exception is the `f1_respondent_id` table, which maps `respondent_id` to the names of the respondents. For that table, we have allowed the most recently reported record to take precedence, overwriting previous mappings if they exist.

Sadly, the FERC Form 1 database is not particularly... relational. The only foreign key relationships that exist map `respondent_id` fields in the individual data tables back to `f1_respondent_id`. In theory, most of the data tables use `report_year`, `respondent_id`, `row_number`, `spplmnt_num` and `report_prd` as a composite primary key (According to this FERC Form 1 database schema from 2015.

In practice, there are several thousand records (out of ~12 million), including some in almost every table, that violate the uniqueness constraint on those primary keys. Since there aren't many meaningful foreign key relationships anyway, rather than dropping the records with non-unique natural composite keys, we chose to preserve all of the records and use surrogate auto-incrementing primary keys in the cloned SQLite database.

5.5 Published Data Packages

We've chosen *tabular data packages* as the main distribution format for PUDL because they:

- are based on a free and open standard that should work on any platform,
- are relatively easy for both humans and computers to understand,

- are easy to archive and distribute,
- provide rich metadata describing their contents,
- do not force users into any particular platform.

We hope this will allow the data to reach the widest possible audience.

See also:

The [Frictionless Data](#) software and specifications, a project of the [Open Knowledge Foundation](#)

5.5.1 Downloading Data Packages

Note: As of `catalystcoop.pudl v0.2.0` we have not yet made our first data release. For the moment you still need to generate your own data packages. However, as soon as v0.2.0 is released, we will start working on a data release, and hope to be able to include the DOI and a link to the Zenodo archive here as of v0.2.1.

Our intent is to automate the creation of a standard bundle of data packages containing all of the currently integrated data. Users who aren't working with Python, or who don't want to set up and run the data processing pipeline themselves will be able to just download and use the data packages directly. Each data release will be issued a DOI, and archived at Zenodo, and may be made available in other ways as well.

Zenodo

Every PUDL software release is automatically [archived and issued a digital object id \(DOI\)](#) by [Zenodo](#) through an integration with [Github](#). The overarching DOI for the entire PUDL project is [10.5281/zenodo.3404014](#), and each release will get its own (versioned) DOI.

On a quarterly basis, we will also upload a standard set of data packages to Zenodo alongside the PUDL release that was used to generate them, and the packages will also be issued citeable DOIs so they can be easily referenced in research and other publications. Our goal is to make replication of any analyses that depend on the released code and published data as easy to replicate as possible.

Other Sites?

Are there other data archiving and access platforms that you'd like to see the pudl data packages published to? If so feel free to [create an issue on Github](#) to let us know about it, and explain what it would add to the project. Other sites we've thought about include:

- [Open EI](#)
- [data.world](#)

5.5.2 Using Data Packages

Once you've downloaded or generated your own tabular data packages you can use them to do analysis on almost any platform. For now, we are primarily using the data packages to populate a local SQLite database.

[Open an issue on Github](#) and let us know if you have another example we can add.

SQLite

If you want to access the data via SQL, we have provided a script that loads a bundle of data packages into a local `sqlite3` database, e.g.:

```
$ datapkg_to_sqlite --pkg_bundle_name pudl-example
```

Python, Pandas, and Jupyter

You can read the datapackages into `pandas.DataFrame` for interactive in-memory use within [JupyterLab](#), or for programmatic use in your own Python modules. Several example Jupyter notebooks are deployed into your PUDL workspace `notebook` directory by the `pudl_setup` script.

Todo: Update `pudl_intro.ipynb` to provide an example of reading the example datapackages directly.

```
$ jupyter lab notebook/pudl_intro.ipynb
```

If you're using Python and need to work with larger-than-memory data, especially the [EPA CEMS Hourly](#) dataset, we recommend checking out [the Dask project](#), which extends the interface to `pandas.DataFrame` objects enabling serialized, parallel and distributed processing tasks. It can also speed up processing for in-memory tasks, especially if you have a powerful system with multiple cores, a solid state disk, and plenty of memory.

The R programming language

Todo: Get someone who uses R to give us an example here... maybe we can get someone from OKFN to do it?

Microsoft Access / Excel

If you'd rather do spreadsheet based analysis, here's how you can pull the datapackages into Microsoft Access and Excel.

Todo: Document process for pulling data packages or datapackage bundles into Microsoft Access / Excel

Other Platforms

Want to submit another example? Check out [the documentation on contributing](#). Wish there was an example here for your favorite data analysis tool, but don't know what it would look like? Feel free to [open a Github issue](#) requesting it.

5.6 Cloud Based Access

As the volume of data integrated into PUDL continues to increase, asking users to either run the processing pipeline themselves, or to download hundreds of gigabytes of data to do their own analyses will become more challenging.

Instead we are working on automatically deploying each data release in cloud computing environments that allow many users to remotely access the same data, as well as computational resources required to work with that data. We hope that this will minimize the technical and administrative overhead associated with using PUDL.

5.6.1 Pangeo

Our focus right now is on the [Pangeo](#) platform, which solves a similar problem for within the Earth science research community. Pangeo uses a [JupyterHub](#) deployment, and includes commonly used scientific software packages and a shared domain specific data repository, which users may access remotely via JupyterLab.

5.6.2 BigQuery

We are also looking at making the published data packages available for live querying by inserting them into Google's [BigQuery](#) data warehouse.

5.6.3 Other Options

Are there other cloud platforms we should consider? Feel free to [create an issue on Github](#) and let us know!

5.7 Data Catalog

Contents

- *Data Catalog*
 - *Available Data*
 - * *EIA Form 860*
 - * *EIA Form 923*
 - * *EPA CEMS Hourly*
 - * *EPA IPM*
 - * *FERC Form 1*
 - *Work in Progress*
 - * *EIA Form 861*
 - * *ISO/RTO LMP*
 - *Future Data*
 - * *EIA Water Usage*
 - * *FERC Form 714*
 - * *FERC EQR*
 - * *MSHA Mines and Production*
 - * *PHMSA Natural Gas Pipelines*
 - * *Transmission and Distribution Systems*

5.7.1 Available Data

Todo: Write up more extensive descriptions of each dataset, what's in them, what the ETL process looks like for each of them, etc. Maybe use this page as an index, with each dataset having its own catalog page. We've got a lot of this information written up elsewhere and should be able to cut-and-paste.

EIA Form 860

Source URL	https://www.eia.gov/electricity/data/eia860/
Source Format	Microsoft Excel (.xls/.xlsx)
Source Years	2001-2017
Size (Download)	127 MB
Size (Uncompressed)	247 MB
PUDL Code	eia860
Years Liberated	2011-2017
Records Liberated	~500,000
Issues	open issues labeled epacems

All of the data reported to the EIA on Form 860 is being pulled into the PUDL database for the years 2011-2017.

We are working on integrating the 2009-2010 EIA 860 data, which has a similar format. This will give us the same coverage in both EIA 860 and EIA 923, which is good since the two datasets are tightly integrated.

Currently we are extending the 2011 EIA 860 data back to 2009 as needed to integrate it with EIA 923.

EIA Form 923

Source URL	https://www.eia.gov/electricity/data/eia923/
Source Format	Microsoft Excel (.xls/.xlsx)
Source Years	2001-2017
Size (Download)	196 MB
Size (Uncompressed)	299 MB
PUDL Code	eia923
Years Liberated	2009-2017
Records Liberated	~2 million
Issues	open issues labeled epacems

Nearly all of EIA Form 923 is being pulled into the PUDL database, for years 2009-2017. Earlier data is available from EIA, but the reporting format for earlier years is substantially different from the present day, and will require more work to integrate. Monthly year to date releases are not yet being integrated.

EPA CEMS Hourly

Source URL	ftp://newftp.epa.gov/dmdnload/emissions/hourly/monthly
Source Format	Comma Separated Value (.csv)
Source Years	1995-2018
Size (Download)	7.6 GB
Size (Uncompressed)	~100 GB
PUDL Code	epacems
Years Liberated	1995-2018
Records Liberated	~1 billion
Issues	open issues labeled epacems

All of the EPA's hourly Continuous Emissions Monitoring System (CEMS) data is available. It is by far the largest dataset in PUDL at the moment, with hourly records for thousands of plants covering decades. Note that the ETL process can easily take all day for the full dataset. PUDL also provides a script that converts the raw EPA CEMS data into Apache Parquet files, which can be read and queried very efficiently from disk. For usage details run:

```
$ epacems_to_parquet --help
```

Thanks to [Karl Dunkle Werner](#) for contributing much of the EPA CEMS Hourly ETL code.

EPA IPM

Source URL	https://www.epa.gov/airmarkets/national-electric-energy-data-system-needs-v6
Source Format	Microsoft Excel (.xlsx)
Source Years	N/A
Size (Download)	14 MB
Size (Uncompressed)	14 MB
PUDL Code	epaipm
Years Liberated	N/A
Records Liberated	~650,000
Issues	open issues labeled epacems

Todo: Get Greg Schivley to write up a description of the EPA IPM dataset.

FERC Form 1

Source URL	https://www.ferc.gov/docs-filing/forms/form-1/data.asp
Source Format	FoxPro Database (.DBC/.DBF)
Source Years	1994-2018
Size (Download)	1.4 GB
Size (Uncompressed)	2.5 GB
PUDL Code	ferc1
Years Liberated	1994-2018 (raw), 2004-2017 (parboiled)
Records Liberated	~12 million (raw), ~270,000 (parboiled)
Issues	open issues labeled

We have integrated a subset of the FERC Form 1 data, mostly pertaining to power plants, their capital & operating expenses, and fuel consumption, for 2004-2017. More work will be required to integrate the rest of the years and data. However we make *all* of the FERC Form 1 data available (7.2 GB of data in 116 tables, going back to 1994) in its raw form via an SQLite database. See *Cloning FERC Form 1* for details.

We continue to improve the integration between the FERC Form 1 plants and the EIA plants and generators, many of which represent the same utility assets. Over time if there's demand we may pull in and clean up additional FERC Form 1 tables.

When we integrate the 2018 FERC Form 1 data, we will also attempt to extend coverage for already integrated tables as far back as 1994.

5.7.2 Work in Progress

Thanks to a grant from the [Alfred P. Sloan Foundation Energy & Environment Program](#), we have support to integrate the following new datasets.

EIA Form 861

Source URL	https://www.eia.gov/electricity/data/eia861/
Source Format	Microsoft Excel (.xls/.xlsx)
Source Years	2001-2017
Size (Download)	–
Size (Uncompressed)	–
PUDL Code	eia861
Years Liberated	–
Records Liberated	–
Issues	open issues labeled epacems

This form includes information about utility demand side management programs, distribution systems, total sales by customer class, net generation, ultimate disposition of power, and other information. This is a smaller dataset (~100s of MB) distributed as Microsoft Excel spreadsheets.

ISO/RTO LMP

Locational marginal electricity pricing information from the various grid operators (e.g. MISO, CAISO, NEISO, PJM, ERCOT...). At high time resolution, with many different delivery nodes, this will be a very large dataset (hundreds of GB). The format for the data is different for each of the ISOs. Physical location of the delivery nodes is not always publicly available.

5.7.3 Future Data

There's a huge variety and quantity of data about the US electric utility system available to the public. The data listed above is just the beginning! Other data we've heard demand for are listed below. If you're interested in using one of them, and would like to add it to PUDL, check out [our contribution guidelines](#). If there are other datasets you think we should be looking at integration, don't hesitate to [open an issue on Github](#) requesting the data and explaining why it would be useful.

EIA Water Usage

[EIA Water](#) records water use by thermal generating stations in the US.

FERC Form 714

[FERC Form 714](#) includes hourly loads, reported by load balancing authorities annually. This is a modestly sized dataset, in the 100s of MB, distributed as Microsoft Excel spreadsheets.

FERC EQR

The [FERC EQR](#) Also known as the Electricity Quarterly Report or Form 920, this dataset includes the details of many transactions between different utilities, and between utilities and merchant generators. It covers ancillary services as well as energy and capacity, time and location of delivery, prices, contract length, etc. It's one of the few public sources of information about renewable energy power purchase agreements (PPAs). This is a large (~100s of GB) dataset, composed of a very large number of relatively clean CSV files, but it requires fuzzy processing to get at some of the interesting and only indirectly reported attributes.

MSHA Mines and Production

The [MSHA Mines & Production](#) dataset describes coal production by mine and operating company, along with statistics about labor productivity and safety. This is a smaller dataset (100s of MB) available as relatively clean and well structured CSV files.

PHMSA Natural Gas Pipelines

The [PHMSA Natural Gas Pipelines](#) dataset, published by the Pipeline and Hazardous Materials Safety Administration (which is part of the US Dept. of Transportation) collects data about the natural gas transmission and distribution system, including their age, length, diameter, materials, and carrying capacity.

Transmission and Distribution Systems

In order to run electricity system operations models and cost optimizations, you need some kind of model of the interconnections between generation and loads. There doesn't appear to be a generally accepted, publicly available set of these network descriptions (yet!).

5.8 Settings Files

Several of the scripts provided as part of PUDL require more arguments than can be easily managed on the command line, and it's useful to preserve a record of how the data processing pipeline was run, so they read their settings from YAML files, examples of which are included in the distribution.

5.8.1 ferc1_to_sqlite

Parameter	Description
<code>ferc1_to_sqlite_year</code>	A single 4-digit year to use as the reference for inferring FERC Form 1 database's structure.
<code>ferc1_to_sqlite_years</code>	A list of years to be included in the cloned FERC Form 1 database. These years must be present in the datastore, and available from FERC (1994 onward).
<code>ferc1_to_sqlite_tables</code>	A list of strings indicating what tables to load. The list of acceptable tables can be found in the example settings file, and corresponds to the values found in the <code>ferc1_dbf2tbl</code> dictionary in <code>pudl.constants</code> .

5.8.2 pudl_etl

The `pudl_etl` script requires a YAML settings file. In the repository this example file lives in `src/pudl/package_data/settings`. This example file (`etl_example.yml`) is deployed onto a user's system in the `pudl_out/settings` directory when the `pudl_setup` script is run. Once this file is in the settings directory, users can modify or copy it and have multiple versions to use with `pudl_etl` for different scenarios.

This settings file is meant to be edited to enable users to set the scope of data that they would like to use in PUDL. Most datasets can be stand-alone data packages. If you only want to use FERC Form 1, you can remove the other data package descriptors, or alter their parameters such that no data would be loaded. The settings are verified early on in the ETL process so if you got something wrong, you should get an assertion error quickly.

While PUDL largely keeps datasets disentangled for ETL purposes (enabling stand-alone ETL) the EPA CEMS and EIA datasets are exceptions. EPA CEMS cannot be loaded without EIA – it relies on IDs that come from EIA 860. Similarly, EIA Forms 860 and 923 are very tightly interdependent. You can load only EIA 860, but the settings verification will automatically add in a few 923 tables that are needed for 860.

The settings verification also removes empty datasets and data packages – the data packages described in the settings file that do not include any years or states, which would generate an empty data package.

Structure of the ETL Settings File

The general structure of the settings file and the names of the keys of the dictionaries should not be changed, but the values of those dictionaries should be edited. There are two high-level elements of the settings file: `pkg_bundle_name` and `pkg_bundle_settings`. The `pkg_bundle_name` will be the directory that the bundle of packages described in the settings file. The elements and structure of the `pkg_bundle_settings` is described below:

```
pkg_bundle_settings
├── name : name of data package
├── title : short title of data package
├── description : longer description of data package
├── datasets
│   ├── dataset name
│   │   ├── dataset etl parameter like states : list of states
│   │   └── dataset etl parameter like years : list of years
│   └── dataset name
│       ├── dataset etl parameter like states : list of states
│       └── dataset etl parameter like years : list of years
└── another data package...
```

The dataset names must not be changed. The dataset names enabled include: `eia`, `ferc1`, `epacems`, `glue` and `epaipm`. Any other dataset name will raise assertion errors.

Dataset ETL parameters (like years, states, tables), will only register if they are a part of its dataset. If you put some FERC etl parameter in an EIA dataset dictionary, FERC will not be loaded as a part of that dataset.

5.9 Contributing to PUDL

PUDL is an open source project that has thus far been supported by a combination of *volunteer efforts and grant funding*. The work is currently being coordinated by the members of [Catalyst Cooperative](#). PUDL is meant to serve a wide variety of public interests including academic research, climate advocacy, data journalism, and public policymaking.

For more on the motivation and history of the project, have a look at [this background info](#). Please also review our *code of conduct*.

5.9.1 How to Get Involved

We welcome just about any kind of contribution to the project. Alone we'll never be able to understand every use case or integrate all the available data. The project will serve the community better if other folks get involved.

There are lots of ways to contribute – it's not all about code!

- [Ask questions on Github](#) using the [issue tracker](#).
- [Suggest new data and features](#) that would be useful.
- [File bug reports](#) on Github.
- Help expand and improve the documentation, or write usage examples.
- Give us feedback on overall usability – what's confusing?
- Tell us a story about how you're using of the data.
- Point us at [interesting publications](#) about energy data, or energy system modeling.
- Cite PUDL if you use the software or data in your own work.
- Share your Jupyter notebooks and other analyses that use PUDL.
- Hire Catalyst to do analysis for your organization based on the PUDL data.
- Contribute code via [pull requests](#). See the [developer setup](#) for more details.

5.9.2 Code of Conduct

We want to make the PUDL project welcoming to contributors with different levels of experience and diverse personal backgrounds. If you're interested in contributing please read our [Code of Conduct](#), which is based on the [Contributor Covenant](#).

5.9.3 We Use Github

Github is the primary platform we use to manage the project, integrate contributions, write and publish documentation, answer user questions, automate testing & deployment, etc. [Signing up for a Github account](#) (even if you don't intend to write code) will allow you to participate in online discussions and track projects that you're interested in.

Ask Questions on Github

Asking (and answering) questions is a valuable contribution!

As noted in [How to support open-source software and stay sane](#) It's much more efficient to ask and answer questions in a public forum because then other users and contributors who are having the same problem can find answers without having to re-ask the same question. The forum we're using is our [Github issues](#).

Even if you feel like you have a basic question, we want you to feel comfortable asking for help in public – we (Catalyst) only recently came to this work as activists and policy wonks – so it’s easy to remember when it all seemed frustrating and alien! Sometimes it still does. We want people to use the software and data to do good things in the world. We want you to be able to access it. Using a public forum also enables the community of users to help each other!

Make Suggestions on GitHub

Don’t hesitate to open an issue with a [feature request](#), or a pointer to energy data that needs liberating, or a reference to documentation that’s out of date, or unclear, or missing. Understanding how people are using the software, and how they would *like* to be using the software is very valuable, and will help us make it more useful and usable.

5.10 Development Setup

If you want to contribute code or documentation directly, you’ll need to create your own fork of the project on Github, and set up some version of the development environment described below, before making pull requests to submit new code, documentation, or examples of use.

Note: If you’re new to git and Github, you may want to check out:

- [The Github Workflow](#)
 - [Collaborative Development Models](#)
 - [Forking a Repository](#)
 - [Cloning a Repository](#)
-

5.10.1 Install Python 3.7

We use [Anaconda](#) or [miniconda](#) to manage our software environments. While using `conda` isn’t strictly required, it does make everything much easier to have everyone on the same platform.

5.10.2 Fork and Clone the PUDL Repository

On the [main page of the PUDL repository](#) you should see a **Fork** button in the upper right hand corner. [Forking the repository](#) makes a copy of it in your personal (or organizational) account on Github that is independent of, but linked to, the original “upstream” project.

Depending on your operating system and the git client you’re using to access Github, the exact cloning process might be different, but if you’re using a UNIX-like terminal, [cloning the repository](#) from your fork will look like this (with your own Github username or organizational name in place of `USERNAME` of course):

```
$ git clone https://github.com/USERNAME/pudl.git
```

This will download the whole history of the project, including the most recent version, and put it in a local directory called `pudl`.

Repository Organization

Inside your newly cloned local repository, you should see the following:

Directory / File	Contents
docs/	Documentation source files for Sphinx and Read The Docs .
src/	Package source code, isolated to avoid unintended imports.
results/	A graveyard of old Jupyter Notebooks and outputs. Ignore!
scripts/	Development scripts not distributed with the package.
test/	Modules for use with PyTest .
environment.yml	File defining the pudl conda environment.
MANIFEST.in	Template describing files included in the python package.
pyproject.toml	Configuration for development tools used with the project.
setup.py	Python build and packaging script.
tox.ini	Configuration for the Tox build and test framework.

5.10.3 Create and activate the pudl-dev conda environment

Inside the newly cloned repository, you should see an `environment.yml` file, which specifies the `pudl-dev` conda environment. You can create that environment locally from within the main repository directory by running:

```
$ conda update conda
$ conda config --set channel_priority strict
$ conda env create --name pudl-dev --file environment.yml
$ conda activate pudl-dev
```

5.10.4 Install PUDL for development

The `catalystcoop.pudl` package isn't part of the `pudl-dev` environment since you're going to be editing it. To install the local version that now exists in your cloned repository using `pip`, from the main repository directory (containing `setup.py`) run:

```
$ pip install --editable ./
```

5.10.5 Install PUDL QA/QC tools

We use automated tools to apply uniform coding style and formatting across the project codebase. This reduces merge conflicts, makes the code easier to read, and helps catch bugs before they are committed. These tools are part of the `pudl` conda environment, and their configuration files are checked into the Github repository, so they should be installed and ready to go if you've cloned the `pudl` repo and are working inside the `pudl` conda environment.

These tools can be run at three different stages in development:

- inside your [text editor](#) or [IDE](#), while you are writing code or documentation,
- before you make a new commit to the repository using Git's [pre-commit hook scripts](#),
- when the *tests are run* – either locally or on a [continuous integration \(CI\)](#) platform (PUDL uses [Travis CI](#)).

See also:

[Real Python Code Quality Tools and Best Practices](#) gives a good overview of available linters and static code analysis tools.

flake8

Flake8 is a popular Python [linting](#) framework, with a large selection of plugins. We use it to run the following checks:

- [PyFlakes](#), which checks Python code for correctness,
- [pycodestyle](#) which checks whether code complies with [PEP 8](#) formatting guidelines,
- [mccabe](#) a tool that measures [code complexity](#) to highlight functions that need to be simplified or reorganized.
- [pydocstyle](#) checks that Python docstrings comply with [PEP 257](#) (via the [flake8-docstrings](#) plugin).
- [pep8-naming](#) checks that variable names comply with Python naming conventions.
- [flake8-builtins](#) checks to make sure you haven't accidentally clobbered any reserved Python names with your own variables.

doc8

Doc8 is a lot like flake8, but for Python documentation written in the reStructuredText format and built by [Sphinx](#). This is the de-facto standard for Python documentation. The doc8 tool checks for syntax errors and other formatting issues in the documentation source files under the `docs/` directory.

autopep8

Instead of just alerting you that there's a style issue in your Python code, [autopep8](#) tries to fix it automatically, applying consistent formatting rules based on [PEP 8](#).

isort

Similarly [isort](#) consistently groups and orders Python import statements in each module.

Python Editors

Many of the tools outlined above can be run automatically in the background while you are writing code or documentation, if you are using an editor that works well with for Python development. A couple of popular options are the free [Atom editor](#) developed by Github, and the less free [Sublime Text editor](#). Both of them have many community maintained addons and plugins.

See also:

[Real Python Guide to Code Editors and IDEs](#)

Catalyst primarily uses the Atom editor, with the following plugins and settings. These plugins require that the tools described above are installed on your system – which is done automatically in the pudl conda environment.

- [atom-beautify](#) set to “beautify on save,” with [autopep8](#) as the beautifier and formatter, and set to “sort imports.”
- [linter](#) the base linter package used by all Atom linters.
- [linter-flake8](#) set to use `.flake8` as the project config file.
- [python-autopep8](#) to actually do the work of tidying up.
- [python-indent](#) to autoindent your code as you write, in accordance with [PEP 8](#).

Git Pre-commit Hooks

Git hooks let you automatically run scripts at various points as you manage your source code. “Pre-commit” hook scripts are run when you try to make a new commit. These scripts can review your code and identify bugs, formatting errors, bad coding habits, and other issues before the code gets checked in. This gives you the opportunity to fix those issues first.

Pretty much all you need to do is enable pre-commit hooks:

```
$ pre-commit install
```

The scripts that run are configured in the `.pre-commit-config.yaml` file.

In addition to `autopep8`, `isort`, `flake8`, and `doc8`, the pre-commit hooks also run `bandit` (a tool for identifying common security issues in Python code) and several other checks that keep you from accidentally committing large binary files, leaving `debugger breakpoints` in your code, forgetting to resolve merge conflicts, and other gotchas that can be hard for humans to catch but are easy for a computer.

Note: If you want to make a pull request, it’s important that all these checks pass – otherwise *the build* will fail, since these same checks are run by the tests on Travis.

See also:

The [pre-commit project](#): A framework for managing and maintaining multi-language pre-commit hooks.

5.10.6 Install and Validate the Data

In order to work on PUDL development, you’ll probably need to have a bunch of the data available locally. Follow the instructions in *Creating a Datastore* to set up a local data management environment and download some data locally, then *run the ETL pipeline to generate some data packages* and use them to populate a local SQLite database with as much PUDL data as you can stand (for development, we typically load all of the available data for `ferc1`, `eia923`, `eia860`, and `epaipm`, datasets, but only a single state’s worth of data for the much larger `epacems` hourly data.)

Using Tox to Validate PUDL

If you’ve done all of the above, you should be able to use `tox` to run our test suite, and perform data validation. For example, to validate the data stored in your PUDL SQLite database, you would simply run:

```
$ tox -v -e validate
```

5.10.7 Running the Tests

We also use `tox` to run PyTest against a packaged and separately installed version of the local repository package. Take a peek inside `tox.ini` to see what test environments are available. To run the same tests that will be run on Travis CI when you make a pull request, you can run:

```
$ tox -v -e travis -- --fast
```

This will run the linters and pre-commit checks on all the code, make sure that the docs can be built by Sphinx, and run the ETL process on a single year of data. The `--fast` is passed through to PyTest by `tox` because it is after the `--`. That test will also attempt to download a year of data into a temporary directory. If you want to skip the download step and use your already downloaded datastore, you can point the tests at it with `--pudl_in=AUTO`:

```
$ tox -v -e travis -- --fast --pudl_in=AUTO
```

Additional details can be found in *Building and Testing PUDL*.

5.10.8 Making a Pull Request

Before you make a pull request, please check that:

- Your code passes all of the Travis tests by running them with `tox`
- You can generate a new complete bundle of data packages, including all the available data (with the exception of `epacems` – all the years of a couple of states is sufficient for testing.)
- Those data packages can be used to populate an SQLite database locally, using the `datapkg_to_sqlite` script.
- The data validation tests can be run against that SQLite database, using `tox -v -e validate` as outlined above.
- If you’ve added new data or substantial new code, please also include new tests and data validation. See the modules under `test` and `test/validate` for examples.

Then you can push the new code to your fork of the PUDL repository on Github, and from there, you can make a Pull Request inviting us to review your code and merge your improvements in with the main repository!

5.11 Building and Testing PUDL

The PUDL Project uses [PyTest](#) to test our code, and [Tox](#) to ensure the tests are run in a controlled environment. We run the tests locally, and on [Travis CI](#).

5.11.1 Test Data

We use the same testing framework to validate the data products being generated by PUDL. This makes running the tests a little more complicated than normal. In addition to specifying what tests should be run, you must specify how much data should be used, and where that data can be found.

Data Quantity:

- For “fast” tests we use the most recent year of data that’s available for all data sources, with the exception of *EPA CEMS Hourly*, for which we only do the most recent year of data for a single state.
- For “full” tests we process all of the data that we expect to work, again with the exception of *EPA CEMS Hourly* for which we do only a single state (across all years).

Data Source:

The tests can use data from three different sources, depending on what you’re testing. They can:

- download a fresh copy of the original data,
- use an existing local datastore skipping the download step, or
- use already processed local data, in the case of post-ETL data validation.

Because [FTP doesn't work on Travis](#), and the [FERC Form 1](#) and [EPA CEMS Hourly](#) data can only be downloaded over FTP, we also keep a small amount of data for those sources in the PUDL Github repository and use it to populate the datastore for continuous integration. We download fresh data for the EIA and other data sources that are available via HTTPS.

5.11.2 Running PyTest

The PyTest suite is organized into two main categories. **ETL** tests and **data validation** tests.

ETL Tests

The ETL tests run the data processing pipeline on either the most recent year of data, or all working years of data. The tests should be marked with `pytest.mark` decorators called `pytest.mark.etl`. Most of the ETL test functions are stored in the `test/etl_test.py` module, but they rely heavily on fixtures defined in `test/conftest.py`. As mentioned above, the data to be used in the ETL tests can come from several different places. You can also specify where the data packages output by the tests should be written.

To run the ETL tests using just the most recent year of data (`--fast`) and download a fresh copy of that data to a temporary location (the default behavior), you would run:

```
$ pytest test/etl_test.py --fast
```

To use an already downloaded copy of the input data, generated a `ferc1` database, in your default *PUDL workspace* (which is specified in `$HOME/.pudl.yml`), you would run:

```
$ pytest test/etl_test.py --fast --pudl_in=AUTO --live_ferc_db=AUTO
```

To specify a particular `pudl_in` directory, containing a data directory and datastore, you would use:

```
$ pytest test/etl_test.py --fast --pudl_in=path/to/pudl_in
```

To change where the output of the ETL pipeline is written, use the `--pudl_out` option. By default it will use a temporary directory created by `pytest`. As with `--pudl_in` you can specify `AUTO` if you want the output to go to your default `pudl_out` (as specified in `$HOME/.pudl.yml`).

```
$ pytest test/etl_test.py --fast --pudl_in=AUTO --pudl_out=my/new/outdir
```

You may also want to consider using `--disable-warnings` to avoid seeing a bunch of clutter from underlying libraries and deprecated uses.

Data Validation Tests

The data validation tests are organized into datasource specific modules under `test/validate`. They test the quality and internal consistency of the data that is output by the PUDL ETL pipeline. Currently they only work on the full dataset, and do not have a `--fast` option. While it is possible to run the full ETL process and output it in a temporary directory, to then be used by the data validation tests, that takes a long time, and you don't get to keep the processed data afterward. Typically we validate outputs that we're hoping to keep around, so we advise running the data validation on an generated PUDL SQLite database.

To point the tests at already processed data, use the `--live_pudl_db` and `--live_ferc_db` options. The `--pudl_in` and `--pudl_out` options work the same as above. E.g.

```
$ pytest --live_pudl_db=AUTO --live_ferc_db=AUTO --pudl_in=AUTO --pudl_out=AUTO test/
↳ validate
```

Data Validation Notebooks

We maintain and test a collection of Jupyter Notebooks that use the same functions as the data validation tests and also produce some visualizations of the data to make it easier to understand what's wrong when validation fails. These notebooks are stored in `test/notebooks` and they can be validated with:

```
$ pytest --nbval-lax test/notebooks
```

The notebooks can only be run when the output of the ETL process is available.

If the data validation tests are failing for some reason, you may want to launch those notebooks in Jupyter to get a better sense of what's going on. They are integrated into the test suite to ensure that they remain functional as the project evolves.

5.11.3 Running Tox

Tox is a system for automating Python packaging and testing processes. When `pytest` is run as described above, it has access to the whole PUDL repository (including files that might not be deployed on a user's system by the packaging script), and it also sees whatever python packages you happen to have installed in your local environment (via `pip` or `conda`) which again, may not be anything like what an end user has on their system when they install *pudl*.

To ensure that we are testing *pudl* as it will be installed for a user who is using `pip` or `conda`, Tox packages up the code as specified in `setup.py`, installs it in a virtual environment, and then runs the same `pytest` tests, but against *that* version of PUDL, giving us much more confidence that it will also work if someone else installs it. The behavior of Tox is controlled by the `tox.ini` file in the main repository directory. It describes several test environments:

- `linters`: Static code analyses that catch syntax errors and style issues.
- `etl`: Run the `pytest` tests in `test/etl_test.py` using the data specified on the command line (see below).
- `validate`: Runs the data validation and output tests and validates the distributed notebooks. Requires existing PUDL outputs.
- `docs`: Builds the documentation using [Sphinx](#) based on the docstrings embedded in our code and any additional resources that we have integrated under the `docs` directory, using the same setup as our documentation on [ReadTheDocs](#)
- `travis`: Runs the tests included in the `linters`, `docs` and `etl` tests.

Todo: Modify the data validation tests to work on a single year of data, so they can be run on Travis and also quickly locally.

Command line arguments like `--fast` and `--pudl_in=AUTO` will be passed in to `pytest` by Tox if you add them after `--` on the command line. E.g. to have Tox run the ETL tests using the most recent year of data, using the data you already have on hand in your local datastore you would do:

```
$ tox -e etl -- --fast --pudl_in=AUTO
```

There are other test environments defined in `tox.ini` – including one for each of the individual linters (`flake8`, `doc8`, `pre-commit`, `bandit`, etc.) which are bundled together into the single `linters` test environment for convenience. There are also `build` and `release` test environments that are used to generate and transmit the *pudl* distribution to the Python Package Index for publication.

To see what each of these Tox environments is actually doing, you can look at the `commands` section for each of them in `tox.ini`.

5.11.4 Generating the Documentation

[Sphinx](#) is a system for semi-automatically generating Python documentation, based on doc strings and other content stored in the `docs` directory. [Read The Docs](#) is a platform that automatically re-runs Sphinx for your project every time you make a commit to Github, and publishes the results online so that you always have up to date docs. It also archives docs for all of your previous releases so folks using them can see how things work for their version of the software, even if it's not the most recent.

Sphinx is tightly integrated with the Python programming language and needs to be able to import and parse the source code to do its job. Thus, it also needs to be able to create an appropriate python environment. This process is controlled by `docs/conf.py`.

However, the resources available on Read The Docs are not as extensive as on Travis, and it can't *really* build many of the scientific libraries we depend on from scratch. Package “mocking” allows us to fake-out the system so that the imports succeed, even if difficult to compile packages like `scipy` aren't really installed.

If you are editing the documentation, and need to regenerate the outputs as you go to see your changes reflected locally, from the main directory of the repository you can run:

```
$ sphinx-build -b html docs docs/_build/html
```

This will only update any files that have been changed since the last time the documentation was generated. If you need to regenerate all of the documentation from scratch, then you should remove the existing outputs first:

```
$ rm -rf docs/_build
$ sphinx-build -b html docs docs/_build/html
```

To run the [doc8](#) reStructuredText linter and re-generate the documentation from scratch, you can use the Tox `docs` test environment:

```
$ tox -e docs
```

Note that this will also attempt to regenerate the `sphinx.autodoc` files in `docs/api` for modules that are meant to be documented, using the `sphinx-apidoc` command – this should catch any new modules or subpackages that are added to the repository, and may result in new files that need to be committed to the Github repository in order for them to show up on Read The Docs.

5.11.5 Python Packaging

In order to distribute a ready-to-use package to others via the Python Package Index and `conda-forge` we need to encapsulate it with some metadata and enumerate its dependencies. There are several files that guide this process.

`setup.py`

The `setup.py` script in the top level of the repository coordinates the packaging process, using `setuptools` which is part of the Python standard library. `setup.py` is really just a single function call, to `setuptools.setup()`, and the parameters of that function are metadata related to the Python package. Most of them are relatively self explanatory – like the name of the package, the license it's being released under, search keywords, etc. – but a few are more arcane:

- `use_scm_version`: Instead of having a hard-coded version that's stored in the repository somewhere, handed off to the packaging script, and often ends up being out of date, pull the version from the source code management (SCM) system, in our case git (and Github). To make a release we will first need to [tag a particular revision](#) in git with a version like `v0.1.0`.

- `python_requires='>=3.7, <3.8.0a0'`: Specifies the version or versions of Python on which the package is expected to run. We require at least Python 3.7, and it's accepted best practice to preclude packages from getting installed on the next major version up, since major versions tend to break things. So we require a version less than Python 3.8.
- `setup_requires=['setuptools_scm']`: What *other* packages need to be installed in order for the packaging script to run? Because we are obtaining the package version from our SCM (git/Github) we need the special package that lets us do that magic, which is named `setuptools_scm`. This automatically generated version number can then be accessed in the package metadata, as is done our top-level `__init__.py` file:

```
__version__ = pkg_resources.get_distribution(__name__).version
```

This is convoluted, but also a currently accepted best practice. The changes to the Python packaging & build system being implemented as a result of [PEP 517](#) and [PEP 518](#) should improve the situation.

- `install_requires`: lists all the other packages that need to be installed before `pudl` can be installed. These are our package dependencies. This list plays a role similar to the `environment.yml` file in the main `pudl` repository, but it depends on `pip` not `conda` – in the packaging system we do not have access to `conda`. It turns out this makes our lives difficult because of the kind of Python packages we depend on. More on this below.
- `extras_require`: a dictionary describing optional packages that can be conditionally installed depending on the expected usage of the install.

Todo: Explain the contents of `extras_require`

- `packages=find_packages('src')`: The `packages` parameter takes a list of all the python packages to be included in the distribution that is being packaged. The `setuptools.find_packages` function automatically searches whatever directories it is given for any packages and all of their subpackages. All of the code we want to distribute to users lives under the `src` directory.
- `package_dir={'': 'src'}`: this tells the packaging to treat any modules or packages found in the `src` directory as part of the `root` package of the distribution. This is a vestigial parameter that pertains to the `distutils` which are the predecessor to `setuptools`... but the system still depends on them deep down inside. In our case, we don't have any modules that aren't part of any package – everything is within `pudl`.
- `include_package_data=True`: This tells the packaging system to include any non-python files that it finds in the directories it has been told to package. In our case this is all the stuff inside `package_data` including example settings files, metadata, glue, etc.
- `entry_points`: This parameter tells the packaging what executable scripts should be installed on the user's system, and which modules/functions implement those scripts.

MANIFEST.in

In addition to generating a version number automatically based on our git repository, `setuptools_scm` pulls every single file tracked by the repository and every other random file sitting in the working repository directory into the distribution. This is... not what we want. `MANIFEST.in` allows us to specify in more detail which files should be included and excluded. Mostly we are just including the python package and supporting data, which exist under the `src/pudl` directory.

pyproject.toml

The adoption of [PEP 517](#) and [PEP 518](#) has opened up the possibility of using build and packaging systems besides `setuptools`. The new system uses `pyproject.toml` to specify the build system requirements. Other tools

related to the project can also store their settings in this file making it easier to see how everything is set up, and avoiding the proliferation of different configuration files for e.g. PyTest, Tox, Flake8, Travis, ReadTheDocs, bandit...

5.12 Data and ETL Design Guidelines

Some technical norms and expectations that we strive to adhere to, and hope that contributors can also follow.

Also, we're all learning as we go – if you have suggestions for best practices we might want to adopt, let us know!

5.12.1 Input vs. Output Data

It's important to differentiate between the original data we're attempting to provide easy access to, and analyses or data products that are derived from that original data. The original data is meant to be archived and re-used as an alternative to other users re-processing the raw data from various public agencies.

5.12.2 Minimize Data Alteration

We are trying to provide a uniform, easy-to-use interface to existing public data. We want to provide access to the original data, insofar as that is possible, while still having it be uniform and easy-to-use. Some alteration is unavoidable and other changes make the data much more usable, but these should be made with care and documentation.

- **Make sure data is available at its full, original resolution.** Don't aggregate the data unnecessarily when it is brought into PUDL. However, creating tools to aggregate it in derived data products is very useful.

Todo: Need fuller enumeration of data alteration / preservation principles.

Examples of Acceptable Changes

- Converting all power plant capacities to MW, or all generation to MWh.
- Assigning uniform NA values.
- Standardizing `datetime` types.
- Re-naming columns to be the same across years and datasets.
- Assigning simple fuel type codes when the original data source uses free-form strings that are not programmatically usable.

Examples of Unacceptable Changes

- Applying an inflation adjustment to a financial variable like fuel cost. There are a variety of possible inflation indices users might want to use, so that transformation should be applied in the output layer that sits on top of the original data.
- Aggregating data that has date/time information associated with it into a time series, when the individual records do not pertain to unique timesteps. For example, the *EIA Form 923* Fuel Receipts and Costs table lists fuel deliveries by month, but each plant might receive several deliveries from the same supplier of the same fuel type in a month – the individual delivery information should be retained.
- Computing heat rates for generators in an original table that contains both fuel heat content and net electricity generation, since the heat rate would be a derived value, and not part of the original data.

5.12.3 Make Tidy Data

The best practices in data organization go by different names in data science, statistics, and database design, but they all try to minimize data duplication and ensure an easy to transform uniform structure that can be used for a wide variety of purposes – at least in the source data (i.e. database tables or the published data packages).

- Each column in a table represents a single, homogeneous variable.
- Each row in a table represents a single observation – i.e. all of the variables reported in that row pertain to the same case/instance of something.
- Don't store the same value in more than one place – each piece of data should have an authoritative source.
- Don't store derived values in the archived data sources.

Reading on Tidy Data

- [Tidy Data](#) A paper on the benefits of organizing data into single variable, homogeneously typed columns, and complete single observation records. Oriented toward the R programming language, but the ideas apply universally to organizing data. (Hadley Wickham, The Journal of Statistical Software, 2014)
- [Good enough practices in scientific computing](#) A whitepaper from the organizers of [Software and Data Carpentry](#) on good habits to ensure your work is reproducible and reusable — both by yourself and others! (Greg Wilson et al., PLOS Computational Biology, 2017)
- [Best practices for scientific computing](#) An earlier version of the above whitepaper aimed at a more technical, data-oriented set of scientific users. (Greg Wilson et al., BLOS Biology, 2014)
- [A Simple Guide to Five Normal Forms](#) A classic 1983 rundown of database normalization. Concise, informal, and understandable, with a few good illustrative examples. Bonus points for the ASCII art.

5.12.4 Use Simple Data Types

The Frictionless Data [TableSchema](#) standard includes a modest selection of data types, which are meant to be very widely usable in other contexts. Make sure that whatever data type you're using is included within that specification, but also be as specific as possible within that collection of options.

This is one aspect of a broader “least common denominator” strategy that is common within the open data. This strategy is also behind our decision to distribute the processed data as CSV files (with metadata stored as JSON). Frictionless Data [makes the case](#) for CSV files in their documentation.

5.12.5 Use Consistent Units

Different data sources often use different units to describe the same type of quantities. Rather than force users to do endless conversions while using the data, we try to convert similar quantities into the same units during ETL. For example, we typically convert all electrical generation to MWh, plant capacities to MW, and heat content to MMBTUs (though, MMBTUs are awful: seriously M=1000 because Roman numerals? So MM is a million, despite the fact that M/Mega is a million in SI. And a BTU is... the amount of energy required to raise the temperature of one an *avoirdupois pound* of water by 1 degree *Fahrenheit*?! What century even is this?).

5.12.6 Silo the ETL Process

It should be possible to run the ETL process on each data source independently, and with any combination of data sources included. This allows users to include only the data need. In some cases like the [EIA 860](#) and [EIA 923](#) data,

two data sources may be so intertwined that keeping them separate doesn't really make sense, but that should be the exception, not the rule.

5.12.7 Separate Data from Glue

The glue that relates different data sources to each other should be applied after or alongside the ETL process, and not as a mandatory part of ETL. This makes it easy to pull individual data sources in and work with them even when the glue isn't working, or doesn't yet exist.

5.12.8 Partition Big Data

Our goal is that users should be able to run the ETL process on a decent laptop. However, some of the utility datasets are hundreds of gigabytes in size (e.g. *EPA CEMS*, *FERC EQR*, *ISO/RTO LMP*). Many users will not need to use the entire dataset for the work they are doing. Allow them to pull in only certain years, or certain states, or other sensible partitions of the data if need be, so that they don't run out of memory or disk space, or have to wait hours while data they don't need is being processed.

5.12.9 Naming Conventions

There are only two hard problems in computer science: caching, naming things, and off-by-one errors.

Use Consistent Names

If two columns in different tables record the same quantity in the same units, give them the same name. That way if they end up in the same dataframe for comparison it's easy to automatically rename them with suffixes indicating where they came from. For example net electricity generation is reported to both *FERC Form 1* and *EIA 923*, so we've named columns `net_generation_mwh` in each of those data sources. Similarly, give non-comparable quantities reported in different data sources **different** column names. This helps make it clear that the quantities are actually different.

Follow Existing Conventions

We are trying to use consistent naming conventions for the data tables, columns, data sources, and functions. Generally speaking PUDL is a collection of subpackages organized by purpose (extract, transform, load, analysis, output, datastore...), containing a module for each data source. Each data source has a short name that is used everywhere throughout the project, composed of the reporting agency and the form number or another identifying abbreviation: `ferc1`, `epacems`, `eia923`, `eia8601`, etc. See the [naming conventions](#) document for more details.

5.12.10 Complete, Continuous Time Series

Most of the data in PUDL are time series, ranging from hourly to annual in resolution.

- **Assume and provide contiguous time series.** Otherwise there are just too many possible combinations of cases to deal with. E.g. don't expect things to work if you pull in data from 2009-2010, and then also from 2016-2018, but not 2011-2015.
- **Assume and provide complete time series.** In data that is indexed by date or time, ensure that it is available as a complete time series, even if some values are missing (and thus NA). Many time series analyses only work when all the timesteps are present.

5.13 Integrating a New Dataset

If you're already working with US energy system data in Python, or have been thinking about doing so, and would like to have the added benefit of access to all the other information that's already part of PUDL, you might consider adding a new data source. That way other people can use the data too, and we can all share the responsibility for ensuring that the code continues to work, and improves over time.

In general the process for adding a new data source looks like this:

1. Add the new data source to the `pudl.workspace.datastore`` module and the `pudl_data` script.
2. Define well normalized data tables for the new data source in the metadata, which is stored in `src/pudl/package_data/meta/datapackage/datapackage.json`.
3. Add a module to the `pudl.extract` subpackage that generates raw dataframes containing the new data source's information from whatever its original format was.
4. Add a module to the `pudl.transform` subpackage that takes those raw dataframes, cleans them up, and re-organizes them to match the new database table definitions.
5. If necessary, add a module to the `pudl.load` subpackage that takes these clean, transformed dataframes and exports them to data packages.
6. If appropriate, create linkages in the table schemas between the tabular resources so they can be used together. Often this means creating some skinny "glue" tables that link one set of unique entity IDs to another.
7. Update the `pudl.etl` module so that it includes your new data source as part of the ETL (Extract, Transform, Load) process, and any necessary code to the `pudl.cli` entrypoint module.
8. Add an output module for the new data source to the `pudl.output` subpackage.
9. Write some unit tests for the new data source, and add them to the `pytest` suite in the `test` directory.

5.13.1 Add dataset to the datastore

Scripts

This means editing the `pudl.workspace.datastore` module and the `pudl_data` script so that they can acquire the data from the reporting agencies, and organize it locally in advance of the ETL process. New data sources should be organized under `data/<agency>/<source>/` e.g. `data/ferc/form1` or `data/eia/form923`. Larger data sources that are available as compressed zipfiles can be left zipped to save local disk space, since `pandas` can read zipfiles directly.

Organization

The exact organization of data within the source directory may vary, but should be as uniform as possible. For data which is compiled annually, we typically make one subdirectory for each year, but some data sources provide all the data in one file for all years (e.g. the MSHA mine info).

User Options

The datastore update script can be run at the command line to pull down new data, or to refresh old data if it's been updated. Someone running the script should be able to specify subsets of the data to pull or refresh – e.g. a set of years, or a set of states – especially in the case of large datasets. In some cases, opening several download connections in parallel may dramatically reduce the time it takes to acquire the data (e.g. pulling down the EPA CEMS dataset over

FTP). The `pudl.constants` module contains several dictionaries which define what years etc. are available for each data source.

Describe Table Metadata

Add table description into *resources* in the the mega-data: the metadata file that contains all of the PUDL table descriptions (`src/pudl/package_data/meta/datapackage/datapackage.json`). The resource descriptions must conform to the [Frictionless Data specifications](#), specifically the specifications for a [tabular data resource](#). The [table schema specification](#) will be particularly helpful.

There is also a dictionary in the megadata called “autoincrement”, which is used for compiling table names that require an auto incremented id column when exporting to a database. This is for tables with no natural primary key. The id column is not required for the datapackages but when exporting to a database, we will read this dictionary in the ``datapkg_to_sqlite`` script to determine which tables need these auto increment id column. Make sure your tables are normalized – see Design Guidelines below.

Extract the data from its original format.

The raw inputs to the extract step should be the pointers to the datastore and any parameters on grabbing the dataset (i.e. the working years, locational constraints if applicable). The outcome of the extract module should be a dictionary of dataframes with keys that correspond to the original datasource table/tab/file name with each row corresponding to one record. These raw dataframes should not be largely altered from their original structures in this step, with the exception of creating records. For example, the EIA 923 often reports a year’s worth of monthly data in one row and the extract step transforms the single row into twelve monthly records. If possible, attempt to keep the dataset in its most compressed format on disk during the extract step. For large data sources stored in zip files (e.g. `epacems`), there is no need to unzip the files as pandas is able to read directly from zipped files. For extracting data from other databases (as opposed to CSV files, spreadsheets, etc.) you may need to populate a live database locally, and read from it (e.g. the FERC Form 1 database, which we clone into postgres from the FoxPro/DBF format used by FERC).

Transform the data into clean normalized dataframes.

The inputs to the transform step should be the dictionary of raw dataframes and any dataset constraints (i.e. working years, tables, and geographical constraints). The output should be a dictionary of transformed dataframes which look exactly like what you want to end up in the database tables. The key of the dictionary should be the name of the database tables as defined in the models. Largely, there is one function per data table. If one database table needs any information such as the index from another table (see `fuel_receipts_costs_eia923` and `coalmine_eia923` for an example), this will require the transform functions to be called in a particular order but the process is largely the same. All the organization of the data into normalized tables happens in the transform step.

During this step, any cleaning of the original data is done. This includes operations like:

- Standardizing units and unit conversions,
- Casting to appropriate data types (string, int, float, date...),
- Conversion to appropriate NA or NaN values for missing data,
- Coding of categorical variables (e.g. fuel type)
- Coding/categorization of freeform strings (e.g. fuel types in FERC Form 1)
- Correction of glaring reporting errors if possible (e.g. when someone reports MWh instead of kWh for net generation, or BTU instead of MMBTU)

Load the data into the datapackages

Each of the dataframes that comes out of the transform step represents a resource that needs to be loaded into the datapackage. Pandas has a native `pandas.DataFrame.to_csv()` method for exporting a dataframe to a CSV file, which is used to output the data to disk.

Because we have not yet taken advantage the new pandas extension arrays, and Python doesn't have a native NA value for integers, just before the dataframes are written to disk we convert any integer NA sentinel values using a little helper function `pudl.helpers.fix_int_na()`.

Glue the new data to existing data

We refer to the links between different data sources as the “glue”. The glue should be able to be thoroughly independent from the ingest of the dataset (there should be no PUDL glue id's in any of the datasource tables and there should be no foreign key relationships from any of the glue tables to the datasource specific tables). These connector keys can be added in the output functions but having them be integral to the database ingestion would make the glue a dependency for adding new datasources, which we want to avoid. The process for adding glue will be very different depending on the datasets you're trying to glue together. The EIA and FERC plants and utilities are currently mapped by hand in a spreadsheet and pulled into tables. The FERC and EIA units ids that will end up living in a glue table will be created through the datazipper. There should be one module in the glue subpackage for each inter-dataset glue (i.e. `ferc1_eia` or `cems_eia`) as well as table definitions in the `models.glue.py` module. If possible, there should be foreign key constraints from the underlying dataset entity tables (i.e. `plants_entity_eia`) to the glue tables so that we do not accidentally store glue that does not refer to the underlying dataset.

Create an output module

The `pudl.output` subpackage compiles interesting information from the database in tabular form for interactive use in dataframes, or for export. Each data source should have its own module in the output subpackage, and within that module there should be a function allowing the output of each of the core tables in the database which come from that data source. These tabular outputs can and should be denormalized, and include additional information a user might commonly want to work with – for example including the names of plants and utilities rather than just their IDs. In addition to those data source specific tabular output modules, there's also `pudl.output.pudltabl.PudlTabl`, a tabular output class. This class can be used to pull and store subsets of the data from the database, and can also use modules within the analysis subpackage to calculate interesting derived quantities, and provide it as a tabular output. See the `pudl.analysis.mcoe` module as an example for how this works.

Write some tests

Test cases need to be created for each new dataset, verifying that the ETL process works, and sanity checking the data itself. This is somewhat different than traditional software testing, since we're not just testing our code – we're also trying to make sure that the data is in good shape. Those exhaustive tests are currently only run locally. See [Building and Testing PUDL](#) for more details.

5.14 Code Standards

- We are trying to keep our own code entirely written in Python.
- PUDL should work on Linux, Mac OS X, or Windows – don't hard code anything that is platform specific, unless you make it work for all platforms.
- Intent is only to support the most recent actively used version or two of Python (Currently Python 3.7).

- Assuming that most if not all users will be using `conda` to manage their Python software environment.
- Make sure the tests run locally, including the linters. See *Building and Testing PUDL* for more information.
- Don't decrease the overall test coverage – if you introduce new code it also needs to be exercised by the tests. See *Building and Testing PUDL* for details.
- Write good docstrings, using the [Google docstring](#) format.
- PUDL should work for use in application development or for interactive analysis (e.g. Jupyter Notebooks).

See also:

- *Development Setup*
- *Building and Testing PUDL*

5.15 Project Management

The people working on PUDL are distributed all over North America. Collaboration takes place online. We make extensive use of Github's project management tools.

5.15.1 Issues and Project Tracking

We use [Github issues](#) to track bugs, enhancements, support requests, and just about any other work that goes into the project. The issues are organized into several different streams of work, using [Github projects](#)

We are happy to accept pull requests that improve our existing code, expand the data that's available via PUDL, and make our documentation more readable and complete. Feel free to report bugs, comment on existing issues, suggest other data sources that might be worth integrating, or ask questions about how to use PUDL if you can't find the answer in our documentation.

5.15.2 Release Management

We are developing and releasing software, but we're also using that software to process and publish data. Our goal is to make the data pipeline as easily and reliably replicable as possible.

Whenever we tag a release on Github, the repository is archived on [Zenodo](#) and issued a DOI. Then the package is uploaded to the Python Package Index for distribution. Our goal is to make a software release at least once a quarter.

Data releases will also be archived on Zenodo, and consist of a software release, a collection of input files, and the resulting data packages. The goal is to make the data package output reproducible given the archived input files and software release, with a single command. Our goal is to make data releases quarterly as well.

5.15.3 User Support

We don't (yet) have funding to do user support, so it's currently all community and volunteer based. In order to ensure that others can find the answers to questions that have already been asked, we try to do all support in public using Github issues.

5.16 Naming Conventions

In the PUDL codebase, we aspire to follow the naming and other conventions detailed in [PEP 8](#).

- Imperative verbs (e.g. connect) should precede the object being acted upon (e.g. connect_db), unless the function returns a simple value (e.g. datadir).
- No duplication of information (e.g. form names).
- lowercase, underscores separate words.
- Helper functions (functions used within a single module only) should be preceded by an underscore.
- When the object is a table, use the full table name (e.g. ingest_fuel_ferc1).
- When dataframe outputs are built from multiple tables, identify the type of information being pulled (e.g. “plants”) and the source of the tables (e.g. eia or ferc1). When outputs are built from a single table, simply use the table name (e.g. boiler_fuel_eia923).

5.16.1 Glossary of Abbreviations

General Abbreviations

Abbreviation	Definition
abbr	abbreviation
assn	association
avg	average (mean)
bbl	barrel (quantity of liquid fuel)
capex	capital expense
corr	correlation
db	database
df & dfs	dataframe & dataframes
dir	directory
epxns	expenses
info	information
mcf	thousand cubic feet (volume of gas)
mmbtu	million British Thermal Units
mw	Megawatt
mwh	Megawatt Hours
num	number
opex	operating expense
pct	percent
ppm	parts per million
ppb	parts per billion
q	(fiscal) quarter
qty	quantity
util & utils	utility & utilities
us	United States
usd	US Dollars

Data Source Specific Abbreviations

Abbreviation	Definition
<code>frc_eia923</code>	Fuel Receipts and Costs (<i>EIA Form 923</i>)
<code>gen_eia923</code>	Generation (<i>EIA Form 923</i>)
<code>gf_eia923</code>	Generation Fuel (<i>EIA Form 923</i>)
<code>gens_eia923</code>	Generators (<i>EIA Form 923</i>)
<code>utils_eia860</code>	Utilities (<i>EIA Form 860</i>)
<code>own_eia860</code>	Ownership (<i>EIA Form 860</i>)

5.16.2 Data Extraction Functions

The lower level namespace uses an imperative verb to identify the action the function performs followed by the object of extraction (e.g. `get_eia860_file`). The upper level namespace identifies the dataset where extraction is occurring.

5.16.3 Output Functions

When dataframe outputs are built from multiple tables, identify the type of information being pulled (e.g. `plants`) and the source of the tables (e.g. `eia` or `ferc1`). When outputs are built from a single table, simply use the table name (e.g. `boiler_fuel_eia923`).

5.16.4 Table Names

See [this article](#) on database naming conventions.

- Table names in snake_case
- The data source or label (e.g. “`plant_id_pudl`”) should follow the thing it is describing

5.16.5 Columns and Field Names

- `total` should come at the beginning of the name (e.g. `total_expns_production`)
- Identifiers should be structured `type_id_source` where `source` is the agency or organization that has assigned the ID. (e.g. `plant_id_eia`)
- The data source or label (e.g. `plant_id_pudl`) should follow the thing it is describing
- Units should be appended to field names where applicable (e.g. `net_generation_mwh`). This includes “per unit” signifiers (e.g. `_pct` for percent, `_ppm` for parts per million, or a generic `_per_unit` when the type of unit varies, as in columns containing a heterogeneous collection of fuels)
- Financial values are assumed to be in US Dollars.
- `_id` indicates the field contains a usually numerical reference to another table, which will not be intelligible without looking up the value in that other table.
- The suffix `_code` indicates the field contains a short abbreviation from a well defined list of values, that probably needs to be looked up if you want to understand what it means.
- The suffix `_type` (e.g. `fuel_type`) indicates a human readable category from a well defined list of values. Whenever possible we try to use these longer descriptive names rather than codes.

- `_name` indicates a longer human readable name, that is likely not well categorized into a small set of acceptable values.
- `_date` indicates the field contains a `Date` object.
- `_datetime` indicates the field contains a full `Datetime` object.
- `_year` indicates the field contains an integer 4-digit year.
- `capacity` refers to nameplate capacity (e.g. `capacity_mw`)— other specific types of capacity are annotated.
- Regardless of what label utilities are given in the original data source (e.g. `operator` in EIA or `respondent` in FERC) we refer to them as `utilities` in PUDL.

5.17 Project Background

The project grew out of frustration with how difficult it is to make use of public data about the US electricity system. In our own climate activism and policy work we found that many non-profit organizations, academic researchers, grass-roots climate activists, energy journalists, smaller businesses, and even members of the public sector were scraping together the same data repeatedly, for one campaign or project at a time, without accumulating many shared, reusable resources. We decided to try and create a platform that would serve the many folks who have a stake in our electricity and climate policies, but may not have the financial resources to obtain commercially integrated data.

Our energy systems affect everyone, and they are changing rapidly. We hope this shared resource will improve the efficiency, quality, accessibility and transparency of research & analysis related to US energy systems.

These ideas have been explored in more depth in papers from Stefan Pfenninger at ETH Zürich and some of the other organizers of the European [Open Energy Modeling Initiative](#) and [Open Power System Data](#) project.

5.17.1 Reading

- [The importance of open data and software: Is energy research lagging behind?](#) (Energy Policy, 2017) Open community modeling frameworks have become common in many scientific disciplines, but not yet in energy. Why is that, and what are the consequences?
- [Opening the black box of energy modeling: Strategies and lessons learned](#) (Energy Strategy Reviews, 2018). A closer look at the benefits available from using shared, open energy system models, including less duplicated effort, more transparency, and better research reproducibility.
- [Open Power System Data: Frictionless Data for Open Power System Modeling](#) (Applied Energy, 2019). An explanation of the motivation and process behind the European OPSD project, which is analogous to our PUDL project, also making use of Frictionless Data Packages.
- [Open Data for Electricity Modeling](#) (BWMi, 2018). A white paper exploring the legal and technical issues surrounding the use of public data for academic energy system modeling. Primarily focused on the EU, but more generally applicable. Based on a BWMi hosted workshop Catalyst took part in during September, 2018.

We also want to bring best practices from the world of software engineering and data science to energy research and advocacy communities. These papers by Greg Wilson and some of the other organizers of the [Software and Data Carpentries](#) are good accessible introductions, aimed primarily at an academic audience:

- [Best practices for scientific computing](#) (PLOS Biology, 2014)
- [Good enough practices in scientific computing](#) (PLOS Computational Biology, 2017)

5.18 Acknowledgments

Thanks to everyone who has helped make this project a reality!

5.18.1 Open Source Contributions

We've been lucky to have some financial support for PUDL over the last 3+ years, but a lot of the work has still been done on a volunteer basis, both by members of Catalyst Cooperative and open source contributors, including:

- [Karl Dunkle Werner](#), a PhD student at UC Berkeley, who did a lot of the integration work for [EPA CEMS Hourly](#).
- [Greg Schivley](#), a recently minted PhD from Carnegie Mellon University, who has pointed us at lots of great open data resources, and integrated the [EPA IPM](#) data.
- [Priya Donti](#), a PhD student at Carnegie Mellon University, for user experience and documentation feedback.
- [Josh Rhodes](#), [Brianna Cote](#), and [Vibrant Clean Energy](#) for submitting Github issues and offering valuable user feedback.

5.18.2 Grant Funding

The [Alfred P. Sloan Foundation Energy & Environment Program](#) has funded one year of concerted work on PUDL aimed at serving the academic research community (May 2019 - April 2020). Many thanks to Will Driscoll for providing feedback on and edits to our Sloan proposal.

The PUDL project has also received support from [The Flora Family Foundation](#) to make the processed data available in [Published Data Packages](#).

5.18.3 Partnerships

We wouldn't be here today without a long list of partners who have helped us along the way.

We're thankful for the opportunity to participate as a pilot project in the [Frictionless Data Reproducible Research](#) program of the [Open Knowledge Foundation](#). Open Knowledge also supported Catalyst to attend [CSVConf,v4](#) in Portland, Oregon.

Our initial 2017 work on PUDL was done as part of a project to enable refinancing and early retirement of uneconomic coal plants, in collaboration with [The Climate Policy Initiative Climate Finance Program](#). (See [these white papers](#) for some of the results.)

The EU based [Open Energy Modeling Initiative](#) and [Open Power System Data](#) projects have taught us a lot about how to wrangle this data for research purposes, and the importance of the legal and licensing dimension of the work.

Thanks to Matt Wassen and Jeff Deal at [Appalachian Voices](#) for giving us a copy of their archive of FERC Form 1 data that's no longer available online.

[Clean Energy Action](#) in Boulder, Colorado gave several of us our first chance to get paid to do energy policy and data analysis work, organizing around Xcel Energy's Colorado coal plants.

The [Rocky Mountain Farmers Union](#) and their [Cooperative Development Center](#) helped us incorporate as a worker cooperative in Colorado, and continue to offer us affordable legal support.

5.18.4 Networking and Moral Support

- Harriet Moyer-Aptekar
- Uday Varadarajan
- Ron Lehr
- Eric Gimon
- Leslie Glustrom
- Bill Stevenson

5.19 The MIT License

Copyright 2017-2019 Catalyst Cooperative and the Climate Policy Initiative

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the “Software”), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

5.20 Catalyst Cooperative Code of Conduct

5.20.1 Our Pledge

In the interest of fostering an open and welcoming environment, we as contributors and maintainers pledge to making participation in our project and our community a harassment-free experience for everyone, regardless of age, body size, disability, ethnicity, gender identity and expression, level of experience, nationality, personal appearance, race, religion, or sexual identity and orientation.

5.20.2 Our Standards

Examples of behavior that contributes to creating a positive environment include:

- Using welcoming and inclusive language
- Being respectful of differing viewpoints and experiences
- Gracefully accepting constructive criticism
- Focusing on what is best for the community
- Showing empathy towards other community members

Examples of unacceptable behavior by participants include:

- The use of sexualized language or imagery and unwelcome sexual attention or advances
- Trolling, insulting/derogatory comments, and personal or political attacks
- Public or private harassment
- Publishing others' private information, such as a physical or electronic address, without explicit permission
- Other conduct which could reasonably be considered inappropriate in a professional setting

5.20.3 Our Responsibilities

Project maintainers are responsible for clarifying the standards of acceptable behavior and are expected to take appropriate and fair corrective action in response to any instances of unacceptable behavior.

Project maintainers have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct, or to ban temporarily or permanently any contributor for other behaviors that they deem inappropriate, threatening, offensive, or harmful.

5.20.4 Scope

This Code of Conduct applies both within project spaces and in public spaces when an individual is representing the project or its community. Examples of representing a project or community include using an official project e-mail address, posting via an official social media account, or acting as an appointed representative at an online or offline event. Representation of a project may be further defined and clarified by project maintainers.

5.20.5 Enforcement

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported by contacting the project team at pudl@catalyst.coop. The project team will review and investigate all complaints, and will respond in a way that it deems appropriate to the circumstances. The project team is obligated to maintain confidentiality with regard to the reporter of an incident. Further details of specific enforcement policies may be posted separately.

Project maintainers who do not follow or enforce the Code of Conduct in good faith may face temporary or permanent repercussions as determined by other members of the project's leadership.

5.20.6 Attribution

This Code of Conduct is adapted from the [Contributor Covenant](http://contributor-covenant.org/version/1/4/), version 1.4, available at <http://contributor-covenant.org/version/1/4/>

5.21 pudl

5.21.1 pudl package

Subpackages

`pudl.analysis` package

Submodules

pudl.analysis.mcoe module

A module with functions to aid generating MCOE.

`pudl.analysis.mcoe.capacity_factor(pudl_out, min_cap_fact=0, max_cap_fact=1.5)`
Calculate the capacity factor for each generator.

Capacity Factor is calculated by using the net generation from eia923 and the nameplate capacity from eia860. The net gen and capacity are pulled into one dataframe, then the dates from that dataframe are pulled out to determine the hours in each period based on the frequency. The number of hours is used in calculating the capacity factor. Then records with capacity factors outside the range specified by `min_cap_fact` and `max_cap_fact` are dropped.

`pudl.analysis.mcoe.fuel_cost(pudl_out)`
Calculate fuel costs per MWh on a per generator basis for MCOE.

Fuel costs are reported on a per-plant basis, but we want to estimate them at the generator level. This is complicated by the fact that some plants have several different types of generators, using different fuels. We have fuel costs broken out by type of fuel (coal, oil, gas), and we know which generators use which fuel based on their `energy_source_code` and reported `prime_mover`. Coal plants use a little bit of natural gas or diesel to get started, but based on our analysis of the “pure” coal plants, this amounts to only a fraction of a percent of their overall fuel consumption on a heat content basis, so we’re ignoring it for now.

For plants whose generators all rely on the same fuel source, we simply attribute the fuel costs proportional to the fuel heat content consumption associated with each generator.

For plants with more than one type of generator energy source, we need to split out the fuel costs according to fuel type – so the gas fuel costs are associated with generators that have `energy_source_code` gas, and the coal fuel costs are associated with the generators that have `energy_source_code` coal.

`pudl.analysis.mcoe.heat_rate_by_gen(pudl_out)`
Convert by-unit heat rate to by-generator, adding fuel type & count.

`pudl.analysis.mcoe.heat_rate_by_unit(pudl_out)`
Calculate heat rates (mmBTU/MWh) within separable generation units.

Assumes a “good” Boiler Generator Association (bga) i.e. one that only contains boilers and generators which have been completely associated at some point in the past.

The BGA dataframe needs to have the following columns: - `report_date` (annual) - `plant_id_eia` - `unit_id_pudl` - `generator_id` - `boiler_id`

The `unit_id` is associated with generation records based on `report_date`, `plant_id_eia`, and `generator_id`. Analogously, the `unit_id` is associated with boiler fuel consumption records based on `report_date`, `plant_id_eia`, and `boiler_id`.

Then the total net generation and fuel consumption per unit per time period are calculated, allowing the calculation of a per unit heat rate. That per unit heat rate is returned in a dataframe containing: - `report_date` - `plant_id_eia` - `unit_id_pudl` - `net_generation_mwh` - `total_heat_content_mmbtu` - `heat_rate_mmbtu_mwh`

`pudl.analysis.mcoe.mcoe(pudl_out, min_heat_rate=5.5, min_fuel_cost_per_mwh=0.0, min_cap_fact=0.0, max_cap_fact=1.5)`
Compile marginal cost of electricity (MCOE) at the generator level.

Use data from EIA 923, EIA 860, and (eventually) FERC Form 1 to estimate the MCOE of individual generating units. The calculation is performed at the time resolution, and for the period indicated by the `pudl_out` object. that is passed in.

Parameters

- **pudl_out** – a PudlTabl object, specifying the time resolution and date range for which the calculations should be performed.
- **min_heat_rate** – lowest plausible heat rate, in mmBTU/MWh. Any MCOE records with lower heat rates are presumed to be invalid, and are discarded before returning.
- **max_cap_fact** (*min_cap_fact*,) – minimum & maximum generator capacity factor. Generator records with a lower capacity factor will be filtered out before returning. This allows the user to exclude generators that aren’t being used enough to have valid.
- **min_fuel_cost_per_mwh** – minimum fuel cost on a per MWh basis that is required for a generator record to be considered valid. For some reason there are now a large number of \$0 fuel cost records, which previously would have been NaN.

Returns a dataframe organized by date and generator, with lots of juicy information about the generators – including fuel cost on a per MWh and MMBTU basis, heat rates, and net generation.

Return type `pandas.DataFrame`

Module contents

Modules providing programmatic analyses that make use of PUDL data.

The `pudl.analysis` subpackage is a collection of modules which implement various systematic analyses using the data compiled by PUDL. Over time this should grow into a rich library of tools that show how the data can be put to use. We may also generate post-analysis datapackages for distribution at some point.

pudl.convert package

Submodules

pudl.convert.datapkg_to_sqlite module

Convert a set of datapackages to SQLite database.

This script will convert a bundle of datapackages into one SQLite database. First, it flattens that datapackages into one datapackage named ‘pudl-all’. Then, it converts that flattened pudl datapackage into a SQLite database.

You will need to give the name of the directory which contains datapackages that you want to flatten and convert. This directory needs to be in your “PUDL_OUT” and “datapackage” directory. To see more info on directory setup see the `pudl_setup` script (`pudl_setup -help` for more details).

```
pudl.convert.datapkg_to_sqlite.main()
```

Convert a set of datapackages to a sqlite database.

```
pudl.convert.datapkg_to_sqlite.parse_command_line(argv)
```

Parse command line arguments. See the `-h` option.

Parameters `argv` (*str*) – Command line arguments, including caller filename.

Returns Dictionary of command line arguments and their parsed values.

Return type `dict`

```
pudl.convert.datapkg_to_sqlite.pkg_to_sqlite_db(pudl_settings, pkg_bundle_name,
                                                pkg_name=None)
```

Turn a data package into a sqlite database.

Parameters

- **pudl_settings** (*dict*) – a dictionary filled with settings that mostly describe paths to various resources and outputs.
- **pkg_name** (*str*) – name of data package. this can be the flattened datapackge (by default named ‘pudl-all’) or any of the sub- datapackages.
- **pkg_bundle_name** (*str*) – the name of the directory where the bundle of datapackages live that you want to convert.

pudl.convert.epacems_to_parquet module

A script for converting the EPA CEMS dataset from gzip to Apache Parquet.

The original EPA CEMS data is available as ~12,000 gzipped CSV files, one for each month for each state, from 1995 to the present. On disk they take up about 7.3 GB of space, compressed. Uncompressed it is closer to 100 GB. That’s too much data to work with in memory.

Apache Parquet is a compressed, columnar datastore format, widely used in Big Data applications. It’s an open standard, and is very fast to read from disk. It works especially well with both Dask dataframes (a parallel / distributed computing extension of Pandas) and Apache Spark (a cloud based Big Data processing pipeline system.)

Since pulling 100 GB of data into postgres takes a long time, and working with that data en masse isn’t particularly pleasant on a laptop, this script can be used to convert the original EPA CEMS data to the more widely usable Apache Parquet format for use with Dask, either on a multi-core workstation or in an interactive cloud computing environment like Pangeo.

For more information on working with these systems check out:

- <https://tomaugspurger.github.io/modern-1-intro>
- <https://dask.pydata.org>
- <https://pangio.io>

```
pudl.convert.epacems_to_parquet.create_cems_schema()
```

Make an explicit Arrow schema for the EPA CEMS data.

Make changes in the types of the generated parquet files by editing this function.

Note that parquet’s internal representation doesn’t use unsigned numbers or 16-bit ints, so just keep things simple here and always use int32 and float32.

Returns An Arrow schema for the EPA CEMS data.

Return type pyarrow.schema

```
pudl.convert.epacems_to_parquet.epacems_to_parquet(epacems_years, epacems_states,
                                                    data_dir, out_dir, pkg_dir,
                                                    compression='snappy', partition_cols=('year', 'state'))
```

Take transformed EPA CEMS dataframes and output them as Parquet files.

We need to do a few additional manipulations of the dataframes after they have been transformed by PUDL to get them ready for output to the Apache Parquet format. Mostly this has to do with ensuring homogeneous data types across all of the dataframes, and downcasting to the most efficient data type possible for each of them. We also add a ‘year’ column so that we can partition the dataset on disk by year as well as state.

Parameters

- **epacems_years** (*list*) – list of years from which we are trying to read CEMs data
- **epacems_states** (*list*) – list of years from which we are trying to read CEMs data

- **data_dir** (*path-like*) – Path to the top directory of the PUDL datastore.
- **out_dir** (*path-like*) – The directory in which to output the Parquet files
- **pkg_dir** (*path-like*) – The directory in which to output...
- **compression** (*type?*) –
- **partition_cols** (*type?*) –

Raises `AssertionError` – Raised if an output directory is not specified.

Todo: Return to

`pudl.convert.epacems_to_parquet.main()`

Convert zipped EPA CEMS Hourly data to Apache Parquet format.

`pudl.convert.epacems_to_parquet.parse_command_line(argv)`

Parse command line arguments. See the -h option.

Parameters `argv` (*str*) – Command line arguments, including caller filename.

Returns Dictionary of command line arguments and their parsed values.

Return type `dict`

`pudl.convert.epacems_to_parquet.year_from_operating_datetime(df)`

Add a 'year' column based on the year in the operating_datetime.

Parameters `df` (*pandas.DataFrame*) – A DataFrame containing EPA CEMS data.

Returns A DataFrame containing EPA CEMS data with a 'year' column.

Return type `pandas.DataFrame`

`pudl.convert.ferc1_to_sqlite` module

A script for cloning the FERC Form 1 database into SQLite.

This script generates a SQLite database that is a clone/mirror of the original FERC Form1 database. We use this cloned database as the starting point for the main PUDL ETL process. The underlying work in the script is being done in `pudl.extract.ferc1`.

`pudl.convert.ferc1_to_sqlite.main()`

Clone the FERC Form 1 FoxPro database into SQLite.

`pudl.convert.ferc1_to_sqlite.parse_command_line(argv)`

Parse command line arguments. See the -h option.

Parameters `argv` (*str*) – Command line arguments, including caller filename.

Returns Dictionary of command line arguments and their parsed values.

Return type `dict`

`pudl.convert.flatten_datapkg` module

This module takes a bundle of datapackages and flattens them.

Because we have enabled the generation of multiple data packages as a part of a data package “bundle”, we need to squish the multiple data packages together in order to put all of the pudl data into one data package. This is especially useful for converting the data package to a SQLite database or any other format.

The module does two main things:

- squish the csv’s together
- squish the metadata (datapackage.json) files together

The CSV squishing is pretty simple and is all being done in `flatten_data_packages_csvs`. We are assuming and enforcing that if two data packages include the same dataset, that dataset has the same ETL parameters (years, tables, states, etc.). The metadata is slightly more complicated to compile because each element of the metadata is structured differently. Most of that work is being done in `flatten_data_package_metadata`.

```
pudl.convert.flatten_datapkg.check_for_matching_parameters(pkg_bundle_dir,
                                                         pkg_name)
```

Check to see if the ETL parameters for datasets are the same across dp’s.

Parameters

- **pkg_bundle_dir** (*path-like*) – the subdirectory where the bundle of data packages live
- **pkg_name** (*str*) – the name you choose for the flattened data package.

```
pudl.convert.flatten_datapkg.compile_data_packages_metadata(pkg_bundle_dir,
                                                           pkg_name='pudl-all')
```

Grab the metadata from each of your dp’s.

Parameters

- **pkg_bundle_dir** (*path-like*) – the subdirectory where the bundle of data packages live
- **pkg_name** (*str*) – the name you choose for the flattened data package.

Returns pkg_descriptor_elements

Return type dict

```
pudl.convert.flatten_datapkg.flatten_data_package_metadata(pkg_bundle_dir,
                                                           pkg_name='pudl-all')
```

Convert a bundle of PULD data package metadata into one file.

Parameters

- **pkg_bundle_dir** (*path-like*) – the subdirectory where the bundle of data packages live
- **pkg_name** (*str*) – the name you choose for the flattened data package.

Returns pkg_descriptor

Return type dict

```
pudl.convert.flatten_datapkg.flatten_data_packages_csvs(pkg_bundle_dir,
                                                         pkg_name='pudl-all')
```

Copy the CSVs into a new data package directory.

Parameters

- **pkg_bundle_dir** (*path-like*) – the subdirectory where the bundle of data packages live

- **pkg_name** (*str*) – the name you choose for the flattened data package.

```
pudl.convert.flatten_datapkg.flatten_pudl_datapackages (pudl_settings,
                                                         pkg_bundle_name,
                                                         pkg_name='pudl-all')
```

Combines a collection of PUDL data packages into one.

Parameters

- **pkg_bundle_name** (*str*) – the name of the subdirectory where the bundle of data packages live. Normally, this name will have been generated in *generate_data_packages*.
- **pudl_settings** (*dict*) – a dictionary filled with settings that mostly describe paths to various resources and outputs.
- **pkg_name** (*str*) – the name you choose for the flattened data package.

Returns a dictionary of the data package validation report.

Return type *dict*

```
pudl.convert.flatten_datapkg.get_all_sources (pkg_descriptor_elements)
```

Grab list of all of the datasets in a data package bundle.

```
pudl.convert.flatten_datapkg.get_same_source_meta (pkg_descriptor_elements, title)
```

Grab the the source metadata of the same dataset from all datapackages.

Module contents

Tools for converting datasets between various formats in bulk.

It's often useful to be able to convert entire datasets in bulk from one format to another, both independent of and within the context of the ETL pipeline. This subpackage collects those tools together in one place.

Currently the tools use a mix of idioms, referring either to a particular dataset and a particular format, or two formats. Some of them read from the original raw data as organized by the *pudl.workspace* package (e.g. *pudl.convert.fercl_to_sqlite* or *pudl.convert.epacems_to_parquet*), and others convert the entire collection of data from an output datapackage into another format (e.g. *pudl.convert.datapackage_to_sqlite*).

pudl.extract package

Submodules

pudl.extract.eia860 module

Retrieve data from EIA Form 860 spreadsheets for analysis.

This modules pulls data from EIA's published Excel spreadsheets.

This code is for use analyzing EIA Form 860 data.

```
pudl.extract.eia860.extract (eia860_years, data_dir)
```

Creates a dictionary of DataFrames containing all the EIA 860 tables.

Parameters

- **eia860_years** (*list*) – a list of data_years
- **data_dir** (*str*) – Top level datastore directory.

Returns A dictionary of EIA 860 pages (keys) and DataFrames (values)

Return type `dict`

`pudl.extract.eia860.get_eia860_column_map(page, year)`

Given a year and EIA860 page, returns info needed to slurp it from Excel.

The format of the EIA860 has changed slightly over the years, and so it is not completely straightforward to pull information from the spreadsheets into our analytical framework. This function looks up a map of the various tabs in the spreadsheet by year and page, and returns the information needed to name the data fields in a standardized way, and pull the right cells from each year & page into our database.

Parameters

- **page** (*str*) – The string label indicating which page of the EIA860 we are attempting to read in. Must be one of the following: - ‘generation_fuel’ - ‘stocks’ - ‘boiler_fuel’ - ‘generator’ - ‘fuel_receipts_costs’ - ‘plant_frame’
- **year** (*int*) – The year that we’re trying to read data for.

Returns

A tuple containing:

- `int`: `sheet_name`, an integer indicating which page in the worksheet the data should be pulled from. 0 is the first page, 1 is the second page, etc. For use by `pandas.read_excel()`
- `int`: `skiprows`, an integer indicating how many rows should be skipped at the top of the sheet being read in, before the header row that contains the strings which will be converted into column names in the dataframe which is created by `pandas.read_excel()`
- `dict`: `column_map`, a dictionary that maps the names of the columns in the year being read in, to the canonical EIA923 column names (i.e. the column names as they are in 2014-2016). This dictionary will be used by `DataFrame.rename()`. The keys are the column names in the dataframe as read from older years, and the values are the canonical column names. All should be stripped of leading and trailing whitespace, converted to lower case, and have internal non-alphanumeric characters replaced with underscores.
- `pd.Index`: `all_columns`, the column Index associated with the column map – it includes all of the columns which might be present in all of the years of data, for use in setting the column index of the raw dataframe which is ultimately extracted, so we can ensure that they all have the same columns, even if we’re only loading a limited number of years.

Return type `tuple`

`pudl.extract.eia860.get_eia860_file(yr, file, data_dir)`

Construct the appropriate path for a given EIA860 Excel file.

Parameters

- **year** (*int*) – The year that we’re trying to read data for.
- **file** (*str*) – A string containing part of the file name for a given EIA 860 file (e.g. ‘Generator’)
- **data_dir** (*str*) – Top level datastore directory.

Returns Path to EIA 860 spreadsheets corresponding to a given year.

Return type `str`

Raises `AssertionError` – If the requested year is not in the list of working years for EIA 860.

```
pudl.extract.eia860.get_eia860_page(page, eia860_xlsx, years=(2011, 2012, 2013, 2014, 2015,
2016, 2017))
```

Reads a table from several years of EIA860 data, returns a DataFrame.

Parameters

- **page** (*str*) – The string label indicating which page of the EIA860 we are attempting to read in. The page argument must be exactly one of the following strings:
 - 'boiler_generator_assn'
 - 'utility'
 - 'plant'
 - 'generator_existing'
 - 'generator_proposed'
 - 'generator_retired'
 - 'ownership'
- **eia860_xlsx** (*pandas.io.excel.ExcelFile*) – xlsx file of EIA Form 860 for input year or years
- **years** (*list*) – The set of years to read into the DataFrame.

Returns A DataFrame of EIA 860 data from selected page, years.

Return type *pandas.DataFrame*

Raises

- **AssertionError** – If the page string is not among the list of recognized EIA 860 page strings.
- **AssertionError** – If the year is not in the list of years that work for EIA 860.

```
pudl.extract.eia860.get_eia860_xlsx(years, filename, data_dir)
```

Read in Excel files to create Excel objects.

Rather than reading in the same Excel files several times, we can just read them each in once (one per year) and use the ExcelFile object to refer back to the data in memory.

Parameters

- **years** (*list*) – The years that we're trying to read data for.
- **filename** (*str*) – ['enviro_assn', 'utilities', 'plants', 'generators']

Returns xlsx file of EIA Form 860 for input year(s)

Return type *pandas.io.excel.ExcelFile*

pudl.extract.eia923 module

Retrieves data from EIA Form 923 spreadsheets for analysis.

This modules pulls data from EIA's published Excel spreadsheets.

This code is for use analyzing EIA Form 923 data. Currently only years 2009-2016 work, as they share nearly identical file formatting.

```
pudl.extract.eia923.extract(eia923_years, data_dir)
```

Creates a dictionary of DataFrames containing all the EIA 923 tables.

Parameters

- **eia860_years** (*list*) – a list of data_years
- **data_dir** (*str*) – Top level datastore directory.

Returns A dictionary containing EIA 860 pages (keys) and DataFrames of data from each page (values)

Return type *dict*

`pudl.extract.eia923.get_eia923_column_map(page, year)`

Given a year and EIA923 page, returns info needed to slurp it from Excel.

The format of the EIA923 has changed slightly over the years, and so it is not completely straightforward to pull information from the spreadsheets into our analytical framework. This function looks up a map of the various tabs in the spreadsheet by year and page, and returns the information needed to name the data fields in a standardized way, and pull the right cells from each year & page into our database.

Parameters

- **page** (*str*) – The string label indicating which page of the EIA923 we are attempting to read in. Must be one of the following: - ‘generation_fuel’ - ‘stocks’ - ‘boiler_fuel’ - ‘generator’ - ‘fuel_receipts_costs’ - ‘plant_frame’
- **year** (*int*) – The year that we’re trying to read data for.

Returns

A tuple containing:

- **int: sheet_name** (*int*): An integer indicating which page in the worksheet the data should be pulled from. 0 is the first page, 1 is the second page, etc. For use by `pandas.read_excel()`
- **int: skiprows**, an integer indicating how many rows should be skipped at the top of the sheet being read in, before the header row that contains the strings which will be converted into column names in the dataframe which is created by `pandas.read_excel()`
- **int: skiprows**, an integer indicating how many rows should be skipped at the top of the sheet being read in, before the header row that contains the strings which will be converted into column names in the dataframe which is created by `pandas.read_excel()`
- **dict: column_map**, a dictionary that maps the names of the columns in the year being read in, to the canonical EIA923 column names (i.e. the column names as they are in 2014-2016). This dictionary will be used by `DataFrame.rename()`. The keys are the column names in the dataframe as read from older years, and the values are the canonical column names. All should be stripped of leading and trailing whitespace, converted to lower case, and have internal non-alphanumeric characters replaced with underscores.

Return type *tuple*

`pudl.extract.eia923.get_eia923_file(yr, data_dir)`

Construct the appropriate path for a given year’s EIA923 Excel file.

Parameters

- **year** (*int*) – The year that we’re trying to read data for.
- **data_dir** (*str*) – Top level datastore directory.

Returns path to EIA 923 spreadsheets corresponding to a given year.

Return type *str*

```
pudl.extract.eia923.get_eia923_page(page, eia923_xlsx, years=(2009, 2010, 2011, 2012, 2013,
2014, 2015, 2016, 2017))
```

Reads a table from given years of EIA923 data, returns a DataFrame.

Parameters

- **page** (*str*) – The string label indicating which page of the EIA923 we are attempting to read in. The page argument must be exactly one of the following strings:
 - 'generation_fuel'
 - 'stocks'
 - 'boiler_fuel'
 - 'generator'
 - 'fuel_receipts_costs'
 - 'plant_frame'
- **eia923_xlsx** (*pandas.io.excel.ExcelFile*) – xlsx file of EIA Form 923 for input year(s)
- **years** (*list*) – The set of years to read into the dataframe.

Returns A dataframe containing the data from the selected page and selected years from EIA 923.

Return type `pandas.DataFrame`

Todo: Convert 2 assert statements to AssertionError

```
pudl.extract.eia923.get_eia923_xlsx(years, data_dir)
```

Reads in Excel files to create Excel objects.

Rather than reading in the same Excel files several times, we can just read them each in once (one per year) and use the ExcelFile object to refer back to the data in memory.

Parameters

- **years** (*list*) – The years that we're trying to read data for.
- **data_dir** (*str*) – Top level datastore directory.

Returns xlsx file of EIA Form 923 for input year(s)

Return type `pandas.io.excel.ExcelFile`

pudl.extract.epacems module

Retrieve data from EPA CEMS hourly zipped CSVs.

This modules pulls data from EPA's published CSV files.

```
pudl.extract.epacems.extract(epacems_years, states, data_dir)
```

Coordinate the extraction of EPA CEMS hourly DataFrames.

Parameters

- **epacems_years** (*list*) – list of years from which we are trying to read CEMS data
- **states** (*list*) – list of states from which we are trying to read CEMS data
- **data_dir** (*path-like*) – Path to the top directory of the PUDL datastore.

Yields *dict* – a dictionary of States (keys) and DataFrames of CEMS data (values)

Todo: This is really slow. Can we do some parallel processing?

`pudl.extract.epacems.read_cems_csv(filename)`

Reads one CEMS CSV file.

Note that some columns are not read. See `epacems_columns_to_ignore`.

Parameters `filename` (*str*) – The name of the file to be read

Returns A DataFrame containing the contents of the CSV file.

Return type `pandas.DataFrame`

pudl.extract.epaipm module

Retrieve data from EPA's Integrated Planning Model (IPM) v6.

Unlike most of the PUDL data sources, IPM is not an annual timeseries. This file assumes that only v6 will be used as an input, so there are a limited number of files.

`pudl.extract.epaipm.create_dfs_epaipm(files, data_dir)`

Makes dictionary of pages (keys) to dataframes (values) for epaipm tabs.

Parameters

- **files** (*list*) – a list of epaipm files
- **data_dir** (*path-like*) – Path to the top directory of the PUDL datastore.

Returns dictionary of pages (key) to dataframes (values)

Return type *dict*

`pudl.extract.epaipm.extract(epaipm_tables, data_dir)`

Extracts data from IPM files.

Parameters

- **epaipm_tables** (*iterable*) – A tuple or list of table names to extract
- **data_dir** (*path-like*) – Path to the top directory of the PUDL datastore.

Returns dictionary of DataFrames with extracted (but not yet transformed) data from each file.

Return type *dict*

`pudl.extract.epaipm.get_epaipm_file(filename, read_file_args, data_dir)`

Reads in files to create dataframes.

No need to use ExcelFile objects with the IPM files because each file is only a single sheet.

Parameters

- **filename** (*str*) – ['single_transmission', 'joint_transmission']
- **read_file_args** (*dict*) – dictionary of arguments for pandas read_*
- **data_dir** (*path-like*) – Path to the top directory of the PUDL datastore.

Returns an xlsx file of EPA IPM data

Return type `pandas.io.excel.ExcelFile`

`pudl.extract.epaipm.get_epaipm_name(file, data_dir)`

Returns the appropriate EPA IPM excel file.

Parameters

- **file** (*str*) – The file that we’re trying to read data for.
- **data_dir** (*path-like*) – Path to the top directory of the PUDL datastore.

Returns The path to EPA IPM spreadsheet.

Return type `str`

pudl.extract.ferc1 module

Tools for extracting data from the FERC Form 1 FoxPro database for use in PUDL.

FERC distributes the annual responses to Form 1 as binary FoxPro database files. This format is no longer widely supported, and so our first challenge in accessing the Form 1 data is to convert it into a modern format. In addition, FERC distributes one database for each year, and these databases are not explicitly linked together. Over time the structure has changed as new tables and fields have been added. In order to be able to use the data to do analyses across many years, we need to bring all of it into a unified structure. However it appears that these changes are only entirely additive – the most recent versions of the DB contain all the tables and fields that existed in earlier versions.

PUDL uses the most recently released year of data as a template, and infers the structure of the FERC Form 1 database based on the strings embedded within the binary files, pulling out the names of tables and their constituent columns. The structure of the database is also informed by information we found on the FERC website, including a mapping between the table names, DBF file names, and the pages of the Form 1 (add link to file, which should distributed with the docs) that the data was gathered from, as well as a diagram of the structure of the database as it existed in 2015 (add link/embed image).

Using this inferred structure PUDL creates an SQLite database mirroring the FERC database using `sqlalchemy`. Then we use a python package called `dbfread` <<https://dbfread.readthedocs.io/en/latest/>> to extract the data from the DBF tables, and insert it virtually unchanged into the SQLite database. However, we do compile a master table of the all the respondent IDs and respondent names, which all the other tables refer to. Unlike the other tables, this table has no `report_year` and so it represents a merge of all the years of data. In the event that the name associated with a given respondent ID has changed over time, we retain the most recently reported name.

This SQLite based compilation of the original FERC Form 1 databases can accommodate all 100+ tables from all the published years of data (beginning in 1994). Including all the data through 2017, the database takes up more than 7GB of disk space. However, almost 90% of that “data” is embedded binary files in two tables. If those tables are excluded, the database is less than 800MB in size.

The process of cloning the FERC Form 1 database(s) is coordinated by a script called `ferc1_to_sqlite` implemented in `pudl.convert.ferc1_to_sqlite` which is controlled by a YAML file. See the example file distributed with the package [here](#) (link!).

Once the cloned SQLite database has been created, we use it as an input into the PUDL ETL pipeline, and we extract a small subset of the available tables for further processing and integration with other data sources like the EIA 860 and EIA 923.

class `pudl.extract.ferc1.FERC1FieldParser` (*table*, *memofile=None*)

Bases: `dbfread.field_parser.FieldParser`

A custom DBF parser to deal with bad FERC Form 1 data types.

parseN (*field*, *data*)

Augments the Numeric DBF parser to account for bad FERC data.

There are a small number of bad entries in the backlog of FERC Form 1 data. They take the form of leading/trailing zeroes or null characters in supposedly numeric fields, and occasionally a naked ‘.’

Accordingly, this custom parser strips leading and trailing zeros and null characters, and replaces a bare ‘.’ character with zero, allowing all these fields to be cast to numeric values.

Parameters

- `()` (*data*) –
- `()` –
- `()` –

Todo: Zane revisit

`pudl.extract.ferc1.accumulated_depreciation(ferc1_meta, ferc1_table, ferc1_years)`

Creates a DataFrame of the fields of accumulated_depreciation_ferc1.

Parameters

- **ferc1_meta** (*sa.MetaData*) – a MetaData object describing the cloned FERC Form 1 database
- **ferc1_table** (*str*) – The name of the FERC 1 database table to read, in this case, the accumulated_depreciation_ferc1.
- **ferc1_years** (*list*) – The range of years from which to read data.

Returns A DataFrame containing all accumulated_depreciation_ferc1 records.

Return type `pandas.DataFrame`

`pudl.extract.ferc1.add_sqlite_table(table_name, sqlite_meta, dbc_map, data_dir, refyear=2017, bad_cols=())`

Adds a new Table to the FERC Form 1 database schema.

Creates a new `sa.Table` object named `table_name` and add it to the database schema contained in `sqlite_meta`. Use the information in the dictionary `dbc_map` to translate between the DBF filenames in the datastore (e.g. `F1_31.DBF`), and the full name of the table in the FoxPro database (e.g. `f1_fuel`) and also between truncated column names extracted from that DBF file, and the full column names extracted from the DBC file. Read the column datatypes out of each DBF file and use them to define the columns in the new Table object.

Parameters

- **table_name** (*str*) – The name of the new table to be added to the database schema.
- **sqlite_meta** (*sa.schema.MetaData*) – The database schema to which the newly defined Table will be added.
- **dbc_map** (*dict*) – A dictionary of dictionaries
- **bad_cols** (*iterable of 2-tuples*) – A list or other iterable containing pairs of strings of the form (table_name, column_name), indicating columns (and their parent tables) which should *not* be cloned into the SQLite database for some reason.

`pudl.extract.ferc1.dbc_filename(year, data_dir)`

Given a year, returns the path to the master FERC Form 1 .DBC file.

Parameters **year** (*int*) – The year that we’re trying to read data for

Returns the file path to the master FERC Form 1 .DBC file for the year

Return type `str`

`pudl.extract.ferc1.dbf2sqlite` (*tables*, *years*, *refyear*, *pudl_settings*, *bad_cols*=())
Clone the FERC Form 1 Databases to SQLite.

Parameters

- **tables** (*iterable*) – What tables should be cloned?
- **years** (*iterable*) – Which years of data should be cloned?
- **refyear** (*int*) – Which database year to use as a template.
- **pudl_settings** (*dict*) – Dictionary containing paths and database URLs used by PUDL.
- **bad_cols** (*iterable of tuples*) – A list of (table, column) pairs indicating columns that should be skipped during the cloning process. Both table and column are strings in this case, the names of their respective entities within the database metadata.

Returns None

```

pudl.extract.ferc1.define_sqlite_db (sqlite_meta, dbc_map, data_dir, tables={ 'f1_106_2009':
'F1_106_2009', 'f1_106a_2009': 'F1_106A_2009',
'f1_106b_2009': 'F1_106B_2009', 'f1_208_elc_dep':
'F1_208_ELC_DEP', 'f1_231_trn_stdycst':
'F1_231_TRN_STDYCST', 'f1_324_elc_expns':
'F1_324_ELC_EXPNS', 'f1_325_elc_cust':
'F1_325_ELC_CUST', 'f1_331_transiso':
'F1_331_TRANSISO', 'f1_338_dep_depl':
'F1_338_DEP_DEPL', 'f1_397_isorto_stl':
'F1_397_ISORTO_STL', 'f1_398_ancl_ps':
'F1_398_ANCL_PS', 'f1_399_mth_peak':
'F1_399_MTH_PEAK', 'f1_400_sys_peak':
'F1_400_SYS_PEAK', 'f1_400a_iso_peak':
'F1_400A_ISO_PEAK', 'f1_429_trans_aff':
'F1_429_TRANS_AFF', 'f1_acb_epda': 'F1_2',
'f1_accumdepr_prvsn': 'F1_3', 'f1_accumdfrrdtaxcr':
'F1_4', 'f1_adit_190_detail': 'F1_5',
'f1_adit_190_notes': 'F1_6', 'f1_adit_amrt_prop':
'F1_7', 'f1_adit_other': 'F1_8', 'f1_adit_other_prop':
'F1_9', 'f1_allowances': 'F1_10', 'f1_allowances_nox':
'F1_ALLOWANCES_NOX', 'f1_audit_log': 'F1_78',
'f1_bal_sheet_cr': 'F1_11', 'f1_capital_stock': 'F1_12',
'f1_cash_flow': 'F1_13', 'f1_cmmn_utlty_p_e':
'F1_14', 'f1_cmpinc_hedge': 'F1_CMPINC_HEDGE',
'f1_cmpinc_hedge_a': 'F1_CMPINC_HEDGE_A',
'f1_co_directors': 'F1_18', 'f1_codes_val': 'F1_76',
'f1_col_lit_tbl': 'F1_79', 'f1_comp_balance_db':
'F1_15', 'f1_construction': 'F1_16',
'f1_control_respdnt': 'F1_17', 'f1_cptl_stk_expns':
'F1_19', 'f1_csscslc_pcsircs': 'F1_20',
'f1_dacs_epda': 'F1_21', 'f1_dscnt_cptl_stk': 'F1_22',
'f1_edcfu_epda': 'F1_23', 'f1_elc_op_mnt_expn':
'F1_27', 'f1_elc_oper_rev_nb': 'F1_26',
'f1_elctrc_erg_acct': 'F1_24', 'f1_elctrc_oper_rev':
'F1_25', 'f1_electric': 'F1_28', 'f1_email':
'F1_EMAIL', 'f1_envrnmntl_expns': 'F1_29',
'f1_envrnmntl_fcly': 'F1_30', 'f1_footnote_data':
'F1_85', 'f1_footnote_tbl': 'F1_87', 'f1_fuel': 'F1_31',
'f1_general_info': 'F1_32', 'f1_gnrt_plant': 'F1_33',
'f1_hydro': 'F1_86', 'f1_ident_attstn': 'F1_88',
'f1_important_chg': 'F1_34', 'f1_incm_stmnt_2':
'F1_35', 'f1_income_stmnt': 'F1_36', 'f1_leased':
'F1_90', 'f1_load_file_names': 'F1_80',
'f1_long_term_debt': 'F1_93', 'f1_misc_dfrddr':
'F1_38', 'f1_miscgen_expnelc': 'F1_37',
'f1_mthly_peak_otpt': 'F1_39', 'f1_mtrl_sppl':
'F1_40', 'f1_nbr_elc_deptemp': 'F1_41',
'f1_nonutility_prop': 'F1_42', 'f1_note_fin_stmnt':
'F1_43', 'f1_nuclear_fuel': 'F1_44', 'f1_officers_co':
'F1_45', 'f1_othr_dfrddr': 'F1_46',
'f1_othr_pd_in_cptl': 'F1_47', 'f1_othr_reg_assets':
'F1_48', 'f1_othr_reg_liab': 'F1_49', 'f1_overhead':
'F1_50', 'f1_pccidica': 'F1_51', 'f1_plant': 'F1_92',
'f1_plant_in_srvce': 'F1_52', 'f1_privilege':
'F1_81', 'f1_pumped_storage': 'F1_53',
'f1_purchased_pwr': 'F1_54', 'f1_rd_demo_actvty':
'F1_59', 'f1_reg_comm_expn': 'F1_56', 'f1_respdnt_control':
'F1_57', 'f1_respondent_id': 'F1_1',
'f1_retained_erng': 'F1_58', 'f1_rg_trn_srv_rev':

```

Defines a FERC Form 1 DB structure in a given SQLAlchemy MetaData object.

Given a template from an existing year of FERC data, and a list of target tables to be cloned, convert that information into table and column names, and data types, stored within a SQLAlchemy MetaData object. Use that MetaData object (which is bound to the SQLite database) to create all the tables to be populated later.

Parameters

- **sqlite_meta** (*sa.MetaData*) – A SQLAlchemy MetaData object which is bound to the FERC Form 1 SQLite database.
- **dbc_map** (*dict of dicts*) – A dictionary of dictionaries, of the kind returned by `get_dbc_map()`, describing the table and column names stored within the FERC Form 1 FoxPro database files.
- **data_dir** (*str*) – A string representing the full path to the top level of the PUDL datastore containing the FERC Form 1 data to be used.
- **tables** (*iterable of strings*) – List or other iterable of FERC database table names that should be included in the database being defined. e.g. 'f1_fuel' and 'f1_steam'
- **refyear** (*integer*) – The year of the FERC Form 1 DB to use as a template for creating the overall multi-year database schema.
- **bad_cols** (*iterable of 2-tuples*) – A list or other iterable containing pairs of strings of the form (table_name, column_name), indicating columns (and their parent tables) which should *not* be cloned into the SQLite database for some reason.

Returns the effects of the function are stored inside `sqlite_meta`

Return type `None`

`pudl.extract.ferc1.drop_tables(engine)`

Drop all FERC Form 1 tables from the SQLite database.

Creates an `sa.schema.MetaData` object reflecting the structure of the database that the passed in `engine` refers to, and uses that schema to drop all existing tables.

Todo: Treat DB connection as a context manager (with/as).

Parameters **engine** (*sa.engine.Engine*) – An SQL Alchemy SQLite database Engine pointing at an existing SQLite database to be deleted.

Returns `None`

```
pudl.extract.ferc1.extract(ferc1_tables=('fuel_ferc1', 'plants_steam_ferc1',
                                           'plants_small_ferc1', 'plants_hydro_ferc1',
                                           'plants_pumped_storage_ferc1', 'plant_in_service_ferc1', 'purchased_power_ferc1',
                                           'accumulated_depreciation_ferc1'),
                           ferc1_years=(2004, 2005, 2006, 2007, 2008, 2009, 2010, 2011,
                                           2012, 2013, 2014, 2015, 2016, 2017), pudl_settings=None)
```

Coordinates the extraction of all FERC Form 1 tables into PUDL.

Parameters

- **ferc1_tables** (*iterable of strings*) – List of the FERC 1 database tables to be loaded into PUDL. These are the names of the tables in the PUDL database, not the FERC Form 1 database.

- **ferc1_years** (*iterable of ints*) – List of years for which FERC Form 1 data should be loaded into PUDL. Note that not all years for which FERC data is available may have been integrated into PUDL yet.

Returns A dictionary of pandas DataFrames, with the names of PUDL database tables as the keys. These are the raw unprocessed dataframes, reflecting the data as it is in the FERC Form 1 DB, for passing off to the data tidying and cleaning fuctions found in the `pudl.transform.ferc1` module.

Return type `dict`

Raises

- **ValueError** – If the year is not in the list of years for which FERC data is available
- **ValueError** – If the year is not in the list of working FERC years
- **ValueError** – If the FERC table requested is not integrated into PUDL
- **AssertionError** – If no `ferc1_meta` tables are found

`pudl.extract.ferc1.fuel(ferc1_meta, ferc1_table, ferc1_years)`
Creates a DataFrame of `f1_fuel` table records with plant names, >0 fuel.

Parameters

- **ferc1_meta** (*sa.MetaData*) – a `MetaData` object describing the cloned FERC Form 1 database
- **ferc1_table** (*str*) – The name of the FERC 1 database table to read, in this case, the `f1_fuel` table.
- **ferc1_years** (*list*) – The range of years from which to read data.

Returns A DataFrame containing `f1_fuel` records that have `plant_names` and non-zero fuel amounts.

Return type `pandas.DataFrame`

`pudl.extract.ferc1.get_dbc_map(year, data_dir, min_length=4)`
Extract names of all tables and fields from a FERC Form 1 DBC file.

Read the DBC file associated with the FERC Form 1 database for the given `year`, and extract all printable strings longer than `min_length`. Select those strings that appear to be database table names, and their associated field for use in re-naming the truncated column names extracted from the corresponding DBF files (those names are limited to having only 10 characters in their names.)

For more info see: <https://github.com/catalyst-cooperative/pudl/issues/288>

Todo: Ideally this routine shouldn't refer to any particular year of data, but right now it depends on the `ferc1_dbf2tbl` dictionary, which was generated from the 2015 Form 1 database.

Parameters

- **year** (*int*) – The year of data from which the database table and column names are to be extracted. Typically this is expected to be the most recently available year of FERC Form 1 data.
- **data_dir** (*str*) – A string representing the full path to the top level of the PUDL dataset containing the FERC Form 1 data to be used.
- **min_length** (*int*) – The minimum number of consecutive printable characters that should be considered a meaningful string and extracted.

Returns a dictionary whose keys are the long table names extracted from the DBC file, and whose values are lists of pairs of values, the first of which is the full name of each field in the table with the same name as the key, and the second of which is the truncated (≤ 10 character) long name of that field as found in the DBF file.

Return type `dict`

`pudl.extract.ferc1.get_dbf_path(table, year, data_dir)`

Given a year and table name, returns the path to its datastore DBF file.

Parameters

- **table** (*string*) – The name of one of the FERC Form 1 data tables. For example ‘f1_fuel’ or ‘f1_steam’
- **year** (*int*) – The year whose data you wish to find.
- **data_dir** (*str*) – A string representing the full path to the top level of the PUDL datastore containing the FERC Form 1 data to be used.

Returns `dbf_path`, a (hopefully) OS independent path including the filename of the DBF file corresponding to the requested year and table name.

Return type `str`

`pudl.extract.ferc1.get_ferc1_meta(pudl_settings)`

Grab the FERC1 db metadata and check for tables.

`pudl.extract.ferc1.get_raw_df(table, dbc_map, data_dir, years=(1994, 1995, 1996, 1997, 1998, 1999, 2000, 2001, 2002, 2003, 2004, 2005, 2006, 2007, 2008, 2009, 2010, 2011, 2012, 2013, 2014, 2015, 2016, 2017, 2018))`

Combines several years of a given FERC Form 1 DBF table into a dataframe.

Parameters

- **table** (*string*) – The name of the FERC Form 1 table from which data is read.
- **dbc_map** (*dict of dicts*) – A dictionary of dictionaries, of the kind returned by `get_dbc_map()`, describing the table and column names stored within the FERC Form 1 FoxPro database files.
- **data_dir** (*str*) – A string representing the full path to the top level of the PUDL datastore containing the FERC Form 1 data to be used.
- **min_length** (*int*) – The minimum number of consecutive printable
- **years** (*list*) – The range of years to be combined into a single DataFrame.

Returns A DataFrame containing several years of FERC Form 1 data for the given table.

Return type `pandas.DataFrame`

`pudl.extract.ferc1.get_strings(filename, min_length=4)`

Extract printable strings from a binary and return them as a generator.

This is meant to emulate the Unix “strings” command, for the purposes of grabbing database table and column names from the F1_PUB.DBC file that is distributed with the FERC Form 1 data.

Parameters

- **filename** (*str*) – the name of the DBC file from which to extract strings
- **min_length** (*int*) – the minimum number of consecutive printable characters that should be considered a meaningful string and extracted.

Yields *str* – result

Todo: Zane revisit

`pudl.extract.ferc1.plant_in_service(ferc1_meta, ferc1_table, ferc1_years)`
 Creates a DataFrame of the fields of `plant_in_service_ferc1`.

Parameters

- **ferc1_meta** (*sa.MetaData*) – a MetaData object describing the cloned FERC Form 1 database
- **ferc1_table** (*str*) – The name of the FERC 1 database table to read, in this case, the `plant_in_service_ferc1` table.
- **ferc1_years** (*list*) – The range of years from which to read data.

Returns A DataFrame containing all `plant_in_service_ferc1` records.

Return type `pandas.DataFrame`

`pudl.extract.ferc1.plants_hydro(ferc1_meta, ferc1_table, ferc1_years)`
 Creates a DataFrame of `f1_hydro` for records that have plant names.

Parameters

- **ferc1_meta** (*sa.MetaData*) – a MetaData object describing the cloned FERC Form 1 database
- **ferc1_table** (*str*) – The name of the FERC 1 database table to read, in this case, the `f1_hydro` table.
- **ferc1_years** (*list*) – The range of years from which to read data.

Returns A DataFrame containing `f1_hydro` records that have plant names.

Return type `pandas.DataFrame`

`pudl.extract.ferc1.plants_pumped_storage(ferc1_meta, ferc1_table, ferc1_years)`
 Creates a DataFrame of `f1_plants_pumped_storage` records with plant names.

Parameters

- **ferc1_meta** (*sa.MetaData*) – a MetaData object describing the cloned FERC Form 1 database
- **ferc1_table** (*str*) – The name of the FERC 1 database table to read, in this case, the `f1_plants_pumped_storage` table.
- **ferc1_years** (*list*) – The range of years from which to read data.

Returns A DataFrame containing `f1_plants_pumped_storage` records that have plant names.

Return type `pandas.DataFrame`

`pudl.extract.ferc1.plants_small(ferc1_meta, ferc1_table, ferc1_years)`
 Creates a DataFrame of `f1_small` for records with minimum data criteria.

Parameters

- **ferc1_meta** (*sa.MetaData*) – a MetaData object describing the cloned FERC Form 1 database
- **ferc1_table** (*str*) – The name of the FERC 1 database table to read, in this case, the `f1_small` table.
- **ferc1_years** (*list*) – The range of years from which to read data.

Returns A DataFrame containing f1_small records that have plant names and non zero demand, generation, operations, maintenance, and fuel costs.

Return type `pandas.DataFrame`

`pudl.extract.ferc1.plants_steam(ferc1_meta, ferc1_table, ferc1_years)`

Creates a DataFrame of f1_steam records with plant names, capacities > 0.

Parameters

- **ferc1_meta** (*sa.MetaData*) – a MetaData object describing the cloned FERC Form 1 database
- **ferc1_table** (*str*) – The name of the FERC 1 database table to read, in this case, the f1_steam table.
- **ferc1_years** (*list*) – The range of years from which to read data.

Returns A DataFrame containing f1_steam records that have plant names and non-zero capacities.

Return type `pandas.DataFrame`

`pudl.extract.ferc1.purchased_power(ferc1_meta, ferc1_table, ferc1_years)`

Creates a DataFrame the fields of purchased_power_ferc1.

Parameters

- **ferc1_meta** (*sa.MetaData*) – a MetaData object describing the cloned FERC Form 1 database
- **ferc1_table** (*str*) – The name of the FERC 1 database table to read, in this case, the purchased_power_ferc1 table.
- **ferc1_years** (*list*) – The range of years from which to read data.

Returns A DataFrame containing all purchased_power_ferc1 records.

Return type `pandas.DataFrame`

`pudl.extract.ferc1.show_dupes(table, dbc_map, years=(1994, 1995, 1996, 1997, 1998, 1999, 2000, 2001, 2002, 2003, 2004, 2005, 2006, 2007, 2008, 2009, 2010, 2011, 2012, 2013, 2014, 2015, 2016, 2017, 2018), pk=['respondent_id', 'report_year', 'report_prd', 'row_number', 'splmnt_num'])`

Identify duplicate primary keys by year within a given FERC Form 1 table.

Parameters

- **table** (*str*) – Name of the original FERC Form 1 table to identify duplicate records in.
- **years** (*iterable*) – a list or other iterable containing the years that should be searched for duplicate records. By default it is all available years of FERC Form 1 data.
- **pk** (*list*) – A list of strings identifying the columns in the FERC Form 1 table that should be treated as a composite primary key. By default this includes: respondent_id, report_year, report_prd, row_number, and splmnt_num.

Returns None

Module contents

Modules implementing the “Extract” step of the PUDL ETL pipeline.

Each module in this subpackage implements data extraction for a single data source from the PUDL *Data Catalog*. This process begins with the original data as retrieved by the `pudl.workspace` subpackage, and ends with a dictionary of “raw” `pandas.DataFrame`’s, that have been minimally altered from the original data, and are ready for normalization and data cleaning by the data source specific modules in the `:mod:`pudl.transform`` subpackage.

pudl.glue package

Submodules

pudl.glue.ferc1_eia module

Extract and transform glue tables between FERC Form 1 and EIA 860/923.

FERC1 and EIA report on the same plants and utilities, but have no embedded connection. We have combed through the FERC and EIA plants and utilities to generate id’s which can connect these datasets. The resulting fields in the PUDL tables are `plant_id_pudl` and `utility_id_pudl`, respectively. This was done by hand in a spreadsheet which is in the `package_data/glue` directory. When mapping plants, we considered a plant a co-located collection of electricity generation equipment. If a coal plant was converted to a natural gas unit, our aim was to consider this the same plant. This module simply reads in the mapping spreadsheet and converts it to a dictionary of dataframes.

Because these mappings were done by hand and for every one of FERC Form 1’s reported plants, we are fairly certain that there are probably some incorrect or incomplete mappings of plants. If you see a `plant_id_pudl` or `utility_id_pudl` mapping that you think is incorrect, poke us on github about it.

A thing to note about using the PUDL id’s is that they can change over time. The spreadsheet uses `MAX({all cells above})` to generate the PUDL id’s for the first instance of every plant or utility, so when an id in the spreadsheet is changed, every id below it is also changed.

Another note about these id’s: these id’s map our definition of plants, which is not the most granular level of plant unit. The generators are typically the smaller, more interesting unit. FERC does not typically report in units (although it sometimes does), but it does often break up gas units from coal units. EIA reports on the generator and boiler level. When trying to use these PUDL id’s, consider the granularity that you desire and the potential implications of using a co-located set of plant infrastructure as an id.

```
pudl.glue.ferc1_eia.glue(ferc1=False, eia=False)
```

Generates a dictionary of dataframes for glue tables between FERC1, EIA.

That data is primarily stored in the `plant_output` and `utility_output` tabs of `package_data/glue/mapping_eia923_ferc1.xlsx` in the repository. There are a total of seven relations described in this data:

- `utilities`: Unique id and name for each utility for use across the PUDL DB.
- `plants`: Unique id and name for each plant for use across the PUDL DB.
- `utilities_eia`: EIA operator ids and names attached to a PUDL utility id.
- `plants_eia`: EIA plant ids and names attached to a PUDL plant id.
- `utilities_ferc`: FERC respondent ids & names attached to a PUDL utility id.
- `plants_ferc`: A combination of FERC plant names and respondent ids, associated with a PUDL plant ID. This is necessary because FERC does not provide plant ids, so the unique plant identifier is a combination of the respondent id and plant name.
- `utility_plant_assn`: An association table which describes which plants have relationships with what utilities. If a record exists in this table then combination of PUDL utility id & PUDL plant id does have an

association of some kind. The nature of that association is somewhat fluid, and more scrutiny will likely be required for use in analysis.

Presently, the ‘glue’ tables are a very basic piece of infrastructure for the PUDL DB, because they contain the primary key fields for utilities and plants in FERC1.

Parameters

- **ferc1** (*bool*) – Are we ingesting FERC Form 1 data?
- **eia** (*bool*) – Are we ingesting EIA data?

Returns a dictionary of glue table DataFrames

Return type `dict`

Module contents

Tools for integrating & reconciling different PUDL datasets with each other.

Many of the datasets integrated by PUDL report related information, but it’s often not easy to programmatically relate the datasets to each other. The glue subpackage provides tools for doing so, making all of the individual datasets more useful, and enabling richer analyses.

In this subpackage there are two basic types of modules:

- those that implement general tools for connecting datasets together (like the `pudl.glue.zipper` module which two tabular datasets based on a set of mutually reported variables with no common IDs), and
- those that implement a connection between two specific datasets (like the `pudl.glue.ferc1_eia` module).

In general we try to enable each dataset to be processed independently, and optionally apply the glue to connect them to each other when both datasets for which glue exists are being processed together.

pudl.load package

Submodules

pudl.load.csv module

A module with functions for loading the pudl database tables.

class `pudl.load.csv.BulkCopy` (*table_name*, *pkg_dir*, *buffer=1073741824*)

Bases: `contextlib.AbstractContextManager`

Accumulate several DataFrames, then COPY FROM pandas to a CSV.

NOTE: You should use this class to load one table at a time. To load different tables, use different instances of BulkCopy.

Parameters

- **table_name** (*str*) – The exact name of the database table which the DataFrame df is going to be used to populate. It will be used both to look up an SQLAlchemy table object in the PUDLBase metadata object, and to name the CSV file.
- **buffer** (*int*) – Size of data to accumulate (in bytes) before actually writing the data into postgresql. (Approximate, because we don’t introspect memory usage ‘deeply’). Default 1 GB.

- **pkg_dir** (*str*) – Path to the directory into which the CSV file should be saved, if it's being kept.

Example

```
>>> with BulkCopy(my_table, my_engine) as p:
    for df in df_generator:
        p.add(df)
```

add(df)

Adds a DataFrame to the accumulated list.

Parameters **df** (*pandas.DataFrame*) – The DataFrame to add to the accumulated list.

Returns None

close()

Output the accumulated tabular data to disk.

Todo: Return to

spill()

Spill the accumulated dataframes into the datapackage.

puhl.load.csv.clean_columns_dump (*table_name, pkg_dir, df*)

Output the cleaned columns to a CSV file.

Parameters

- **table_name** (*str*) –
- **pkg_dir** (*path-like*) –
- **df** (*pandas.DataFrame*) –

Returns None

Todo: Incomplete Docstring

puhl.load.csv.csv_dump (*df, table_name, keep_index, pkg_dir*)

Writes a dataframe to CSV and loads it into postgresql using COPY FROM.

The fastest way to load a bunch of records is using the database's native text file copy function. This function dumps a given dataframe out to a CSV file, and then loads it into the specified table using a sqlalchemy wrapper around the postgresql COPY FROM command, called `postgres_copy`.

Note that this creates an additional in-memory representation of the data, which takes slightly less memory than the DataFrame itself.

Parameters

- **df** (*pandas.DataFrame*) – The DataFrame which is to be dumped to CSV and loaded into the database. All DataFrame columns must have exactly the same names as the database fields they are meant to populate, and all column data types must be directly compatible with the database fields they are meant to populate. Do any cleanup before you call this function.

- **table_name** (*str*) – The exact name of the database table which the DataFrame df is going to be used to populate. It will be used both to look up an SQLAlchemy table object in the PUDLBase metadata object, and to name the CSV file.
- **keep_index** (*bool*) – Should the output CSV file contain an index?
- **pkg_dir** (*path-like*) – Path to the directory into which the CSV file should be saved, if it's being kept.

Returns None

```

pudl.load.csv.dict_dump(transformed_dfs, data_source, need_fix_inting={'coalmine_eia923':
('mine_id_msha', 'county_id_fips'), 'fuel_receipts_costs_eia923':
('mine_id_pudl', ), 'generation_fuel_eia923': ('nuclear_unit_id', ),
'generators_eia860': ('turbines_num', ), 'hourly_emissions_epacems':
('facility_id', 'unit_id_epa'), 'plants_eia860': ('utility_id_eia', ),
'plants_entity_eia': ('zip_code', ), 'plants_hydro_ferc1': ('construction_year', 'installation_year'),
'plants_pumped_storage_ferc1': ('construction_year', 'installation_year'), 'plants_small_ferc1': ('construction_year', 'ferc_license_id'),
'plants_steam_ferc1': ('construction_year', 'installation_year')}, pkg_dir=")

```

Wrapper for _csv_dump for each data source.

Parameters

- **transformed_dfs** (*dict*) – A dictionary of DataFrame objects in which tables from datasets (keys) correspond to normalized DataFrames of values from that table (values)
- **datasource** (*str*) – The name of the datasource we are working with.
- **need_fix_inting** (*dict*) – A dictionary containing table names (keys) and column names for each table that need their null values cleaned up (values).
- **pkg_dir** (*path-like*) – Path to the directory into which the CSV file should be saved, if it's being kept.

Returns None

pudl.load.metadata module

Make me metadata!!!.

Lists of dictionaries of dictionaries of lists, forever. This module enables the generation and use of the metadata for tabular data packages. This module also saves and validates the datapackage once the metadata is compiled. The intended use of the module is to use it *after* generating the CSV's via *etl.py*.

On a basic level, based on the settings in the pkg_settings, tables and sources associated with a data package, we are compiling information about the data package. For the table metadata, we are pulling from the megadata (*pudl/package_data/meta/datapackage/datapackage.json*). Most of the other elements of the metadata is regenerated.

For most tables, this is a relatively straightforward process, but we are attempting to enable partitioning of tables (storing parts of a table in individual CSVs). These partitioned tables are parts of a “group” which can be read by frictionlessdata tools as one table. At each step the process, this module needs to know whether to deal with the full partitioned table names or the cononical table name.

```

pudl.load.metadata.compile_partitions(pkg_settings)

```

Pull out the partitions from data package settings.

Parameters **pkg_settings** (*dict*) – a dictionary containing package settings containing top level elements of the data package JSON descriptor specific to the data package

Returns**Return type** `dict`

```
pudl.load.metadata.data_sources_from_tables_pkg(table_names, testing=False)
```

Look up data sources based on a list of PUDL DB tables.

Parameters

- **tables_names** (*iterable*) – a list of names of ‘seed’ tables, whose dependencies we are seeking to find.
- **testing** (*bool*) – Connected to the test database (True) or live PUDL database (False)?

Returns The set of data sources for the list of PUDL table names.

Return type `set`

```
pudl.load.metadata.generate_metadata(pkg_settings, tables, pkg_dir, uuid_pkgs='2eafa585-0053-49d0-acae-34d1b90fe494')
```

Generate metadata for package tables and validate package.

The metadata for this package is compiled from the `pkg_settings` and from the “megadata”, which is a json file containing the schema for all of the possible pudl tables. Given a set of tables, this function compiles metadata and validates the metadata and the package. This function assumes datapackage CSVs have already been generated.

See Frictionless Data for the tabular data package specification: <http://frictionlessdata.io/specs/tabular-data-package/>

Parameters

- **pkg_settings** (*dict*) – a dictionary containing package settings containing top level elements of the data package JSON descriptor specific to the data package including: * name: short package name e.g. pudl-eia923, ferc1-test, cems_pkg * title: One line human readable description. * description: A paragraph long description. * keywords: For search purposes.
- **tables** (*list*) – a list of tables that are included in this data package.
- **pkg_dir** (*path-like*) – The location of the directory for this package. The data package directory will be a subdirectory in the `datapackage_dir` directory, with the name of the package as the name of the subdirectory.
- **uuid_pkgs** –

Todo: Return to (uuid_pkgs)

Returns a datapackage. See frictionlessdata specs. dict: a validation dictionary containing validity of package and any errors that were generated during packaing.

Return type `datapackage.package.Package`

```
pudl.load.metadata.get_autoincrement_columns(unpartitioned_tables)
```

Grab the autoincrement columns for pkg tables.

```
pudl.load.metadata.get_dependent_tables_from_list_pkg(table_names, testing=False)
```

Given a list of tables, find all the other tables they depend on.

Iterate over a list of input tables, adding them and all of their dependent tables to a set, and return that set. Useful for determining which tables need to be exported together to yield a self-contained subset of the PUDL database.

Parameters

- **table_names** (*iterable*) – a list of names of ‘seed’ tables, whose dependencies we are seeking to find.
- **testing** (*bool*) – Connected to the test database (True) or live PUDL database (False)?

Returns The set of all the tables which any of the input tables depends on, via ForeignKey constraints.

Return type `all_the_tables` (*set*)

```
pudl.load.metadata.get_dependent_tables_pkg(table_name, fk_relash)
```

For a given table, get the list of all the other tables it depends on.

Parameters

- **table_name** (*str*) – The table whose dependencies we are looking for.
- **()** (*fk_relash*) –

Todo: Incomplete docstring.

Returns the set of all the tables the specified table depends upon.

Return type *set*

```
pudl.load.metadata.get_foreign_key_relash_from_pkg(pkg_json)
```

Generate a dictionary of foreign key relationships from pkging metadata.

This function helps us pull all of the foreign key relationships of all of the tables in the metadata.

Parameters **datapackage_json_path** (*path-like*) – Path to the datapackage.json containing the schema from which the foreign key relationships will be read

Returns list of foreign key tables

Return type *dict*

```
pudl.load.metadata.get_repartitioned_tables(tables, partitions, pkg_settings)
```

Get the re-partitioned tables.

Parameters

- **tables** (*list*) – a list of tables that are included in this data package.
- **partitions** (*dict*) –
- **pkg_settings** (*dict*) – a dictionary containing package settings containing top level elements of the data package JSON descriptor specific to the data package.

Returns list of tables including full groups of

Return type *list*

```
pudl.load.metadata.get_source_metadata(data_sources, pkg_settings)
```

Grab sources for metadata.

```
pudl.load.metadata.get_tabular_data_resource(table_name, pkg_dir, partitions=False)
```

Create a Tabular Data Resource descriptor for a PUDL table.

Based on the information in the database, and some additional metadata this function will generate a valid Tabular Data Resource descriptor, according to the Frictionless Data specification, which can be found here: <https://frictionlessdata.io/specs/tabular-data-resource/>

Parameters

- **table_name** (*string*) – table name for which you want to generate a Tabular Data Resource descriptor
- **pkg_dir** (*path-like*) – The location of the directory for this package. The data package directory will be a subdirectory in the *datapackage_dir* directory, with the name of the package as the name of the subdirectory.

Returns A JSON object containing key information about the selected table

Return type Tabular Data Resource descriptor

```
pudl.load.metadata.get_unpartitioned_tables (tables, pkg_settings)
```

Get the tables w/out the partitions.

Because the partitioning key will always be the name of the table without whatever element the table is being partitioned by, we can assume the names of all of the un-partitioned tables to get a list of tables that is easier to work with.

Parameters

- **tables** (*iterable*) – list of tables that are included in this datapackage.
- **pkg_settings** (*dictionary*) –

Returns tables_unpartitioned is a set of un-partitioned tables

Return type iterable

```
pudl.load.metadata.hash_csv (csv_path)
```

Calculates a SHA-256 hash of the CSV file for data integrity checking.

Parameters **csv_path** (*path-like*) – Path the CSV file to hash.

Returns the hexdigest of the hash, with a 'sha256:' prefix.

Return type *str*

```
pudl.load.metadata.package_files_from_table (table, pkg_settings)
```

Determine which files should exist in a package cooresponding to a table.

We want to convert the datapackage tables and any information about package partitioning into a list of expected files. For each table that is partitioned, we want to add the partitions to the end of the table name.

```
pudl.load.metadata.prep_pkg_bundle_directory (pudl_settings, pkg_bundle_name, clobber=False)
```

Create (or delete and create) data package directory.

Parameters

- **pudl_settings** (*dict*) – a dictionary filled with settings that mostly describe paths to various resources and outputs.
- **debug** (*bool*) – If True, return a dictionary with package names (keys) and a list with the data package metadata and report (values).
- **pkg_bundle_name** (*string*) – name of directory you want the bundle of data packages to live. If this is set to None, the name will be defaulted to be the pudl package version.

Returns path-like

```
pudl.load.metadata.pull_resource_from_megadata (table_name)
```

Read a single data resource from the PUDL metadata library.

Parameters **table_name** (*str*) – the name of the table / data resource whose JSON descriptor we are reading.

Returns a Tabular Data Resource Descriptor, as a JSON object.

Return type json

Raises **ValueError** – If table_name is not found exactly one time in the PUDL metadata library.

`pudl.load.metadata.test_file_consistency(tables, pkg_settings, pkg_dir)`

Test the consistency of tables for packaging.

The purpose of this function is to test that we have the correct list of tables. There are three different ways we could determine which tables are being dumped into packages: a list of the tables being generated through the ETL functions, the list of dependent tables and the list of CSVs in package directory.

Currently, this function is supposed to be fed the ETL function tables which are tested against the CSVs present in the package directory.

Parameters

- **pkg_name** (*string*) – the name of the data package.
- **tables** (*list*) – a list of table names to be tested.
- **pkg_dir** (*path-like*) – the directory in which to check the consistency of table files

Raises **AssertionError** – If the tables in the CSVs and the ETL tables are not exactly the same list of tables.

Todo: Determine what to do with the dependent tables check.

`pudl.load.metadata.validate_save_pkg(pkg_descriptor, pkg_dir)`

Validate a data package descriptor and save it to a json file.

Parameters

- **pkg_descriptor** (*dict*) –
- **pkg_dir** (*path-like*) –

Returns report

Module contents

Tools for handling the load set in pudl ETL.

pudl.output package

Submodules

pudl.output.eia860 module

Functions for pulling data primarily from the EIA's Form 860.

`pudl.output.eia860.boiler_generator_assn_eia860(pudl_engine, pt, start_date=None, end_date=None)`

Pull all fields from the EIA 860 boiler generator association table.

Parameters

- **pudl_engine** (*sqlalchemy.engine.Engine*) – SQLAlchemy connection engine for the PUDL DB.
- **pt** (*ImmutableDict*) – a sqlalchemy metadata dictionary of pudl tables
- **start_date** (*date*) – Date to begin retrieving EIA 860 data.
- **end_date** (*date*) – Date to end retrieving EIA 860 data.

Returns A DataFrame containing all the fields from the EIA 860 boiler generator association table.

Return type `pandas.DataFrame`

`pudl.output.eia860.generators_eia860(pudl_engine, pt, start_date=None, end_date=None)`

Pull all fields reported in the generators_eia860 table.

Merge in other useful fields including the latitude & longitude of the plant that the generators are part of, canonical plant & operator names and the PUDL IDs of the plant and operator, for merging with other PUDL data sources.

Fill in data for adjacent years if requested, but never fill in earlier than the earliest working year of data for EIA923, and never add more than one year on after the reported data (since there should at most be a one year lag between EIA923 and EIA860 reporting)

Parameters

- **pudl_engine** (*sqlalchemy.engine.Engine*) – SQLAlchemy connection engine for the PUDL DB.
- **pt** (*ImmutableDict*) – a sqlalchemy metadata dictionary of pudl tables
- **start_date** (*date*) – the earliest EIA 860 data to retrieve or synthesize
- **end_date** (*date*) – the latest EIA 860 data to retrieve or synthesize

Returns A DataFrame containing all the fields of the EIA 860 Generators table.

Return type `pandas.DataFrame`

`pudl.output.eia860.ownership_eia860(pudl_engine, pt, start_date=None, end_date=None)`

Pull a useful set of fields related to ownership_eia860 table.

Parameters

- **pudl_engine** (*sqlalchemy.engine.Engine*) – SQLAlchemy connection engine for the PUDL DB.
- **pt** (*ImmutableDict*) – a sqlalchemy metadata dictionary of pudl tables
- **start_date** (*date*) – date of the earliest data to retrieve
- **end_date** (*date*) – date of the latest data to retrieve

Returns A DataFrame containing a useful set of fields related to the EIA 860 Ownership table.

Return type `pandas.DataFrame`

`pudl.output.eia860.plants_eia860(pudl_engine, pt, start_date=None, end_date=None)`

Pulls all fields from the EIA Plants tables.

Parameters

- **pudl_engine** (*sqlalchemy.engine.Engine*) – SQLAlchemy connection engine for the PUDL DB.
- **pt** (*ImmutableDict*) – a sqlalchemy metadata dictionary of pudl tables
- **start_date** (*date*) – Date to begin retrieving EIA 860 data.

- **end_date** (*date*) – Date to end retrieving EIA 860 data.

Returns A DataFrame containing all the fields of the EIA 860 Plants table.

Return type `pandas.DataFrame`

`pudl.output.eia860.plants_utils_eia860(pudl_engine, pt, start_date=None, end_date=None)`

Create a dataframe of plant and utility IDs and names from EIA 860.

Returns a pandas dataframe with the following columns: - report_date (in which data was reported) - plant_name (from EIA entity) - plant_id_eia (from EIA entity) - plant_id_pudl - utility_id_eia (from EIA860) - utility_name (from EIA860) - utility_id_pudl

Note: EIA 860 data has only been integrated for 2011-2016. If earlier or later years are requested, they will be filled in with data from the first or last years.

Parameters

- **pudl_engine** (`sqlalchemy.engine.Engine`) – SQLAlchemy connection engine for the PUDL DB.
- **pt** (`sqlalchemy.util._collections.immutabledict`) – a sqlalchemy meta-data dictionary of pudl tables
- **start_date** (*date*) – Date to begin retrieving EIA 860 data.
- **end_date** (*date*) – Date to end retrieving EIA 860 data.

Returns A DataFrame containing plant and utility IDs and names from EIA 860.

Return type `pandas.DataFrame`

`pudl.output.eia860.utilities_eia860(pudl_engine, pt, start_date=None, end_date=None)`

Pulls all fields from the EIA860 Utilities table.

Parameters

- **pudl_engine** (`sqlalchemy.engine.Engine`) – SQLAlchemy connection engine for the PUDL DB.
- **pt** (`sqlalchemy.util._collections.immutabledict`) – a sqlalchemy meta-data dictionary of pudl tables
- **start_date** (*date*) – Date to begin retrieving EIA 860 data.
- **end_date** (*date*) – Date to end retrieving EIA 860 data.

Returns A DataFrame containing all the fields of the EIA 860 Utilities table.

Return type `pandas.DataFrame`

pudl.output.eia923 module

Functions for pulling EIA 923 data out of the PUDL DB.

`pudl.output.eia923.boiler_fuel_eia923(pudl_engine, pt, freq=None, start_date=None, end_date=None)`

Pull records from the boiler_fuel_eia923 table in a given data range.

Optionally, aggregate the records over some timescale – monthly, yearly, quarterly, etc. as well as by fuel type within a plant.

If the records are not being aggregated, all of the database fields are available. If they're being aggregated, then we preserve the following fields. Per-unit values are re-calculated based on the aggregated totals. Totals are summed across whatever time range is being used, within a given

plant and fuel type. * fuel_consumed_units (sum) * fuel_mmbtu_per_unit (weighted average) * total_heat_content_mmbtu (sum) * sulfur_content_pct (weighted average) * ash_content_pct (weighted average)

In addition, plant and utility names and IDs are pulled in from the EIA 860 tables.

Parameters

- **pudl_engine** (*sqlalchemy.engine.Engine*) – SQLAlchemy connection engine for the PUDL DB.
- **pt** (*ImmutableDict*) – a sqlalchemy metadata dictionary of pudl tables
- **freq** (*str*) – a pandas timeseries offset alias. The original data is reported monthly, so the best time frequencies to use here are probably month start (freq='MS') and year start (freq='YS').
- **start_date** (*date-like*) – date-like object, including a string of the form 'YYYY-MM-DD' which will be used to specify the date range of records to be pulled. Dates are inclusive.
- **end_date** (*date-like*) – date-like object, including a string of the form 'YYYY-MM-DD' which will be used to specify the date range of records to be pulled. Dates are inclusive.

Returns A DataFrame containing all records from the EIA 923 Boiler Fuel table.

Return type `pandas.DataFrame`

```
pudl.output.eia923.fuel_receipts_costs_eia923(pudl_engine, pt, freq=None,
                                              start_date=None, end_date=None)
```

Pull records from fuel_receipts_costs_eia923 table in given date range.

Optionally, aggregate the records at a monthly or longer timescale, as well as by fuel type within a plant, by setting freq to something other than the default None value.

If the records are not being aggregated, then all of the fields found in the PUDL database are available. If they are being aggregated, then the following fields are preserved, and appropriately summed or re-calculated based on the specified aggregation. In both cases, new total values are calculated, for total fuel heat content and total fuel cost. * plant_id_eia * report_date * fuel_type_code_pudl (formerly energy_source_simple) * fuel_qty_units (sum) * fuel_cost_per_mmbtu (weighted average) * total_fuel_cost (sum) * total_heat_content_mmbtu (sum) * heat_content_mmbtu_per_unit (weighted average) * sulfur_content_pct (weighted average) * ash_content_pct (weighted average) * moisture_content_pct (weighted average) * mercury_content_ppm (weighted average) * chlorine_content_ppm (weighted average)

In addition, plant and utility names and IDs are pulled in from the EIA 860 tables.

Parameters

- **pudl_engine** (*sqlalchemy.engine.Engine*) – SQLAlchemy connection engine for the PUDL DB.
- **pt** (*ImmutableDict*) – a sqlalchemy metadata dictionary of pudl tables
- **freq** (*str*) – a pandas timeseries offset alias. The original data is reported monthly, so the best time frequencies to use here are probably month start (freq='MS') and year start (freq='YS').
- **start_date** (*date-like*) – date-like object, including a string of the form 'YYYY-MM-DD' which will be used to specify the date range of records to be pulled. Dates are inclusive.
- **end_date** (*date-like*) – date-like object, including a string of the form 'YYYY-MM-DD' which will be used to specify the date range of records to be pulled. Dates are inclusive.

Returns A DataFrame containing all records from the EIA 923 Fuel Receipts and Costs table.

Return type `pandas.DataFrame`

```
pudl.output.eia923.generation_eia923(pudl_engine, pt, freq=None, start_date=None,
                                     end_date=None)
```

Pull records from the boiler_fuel_eia923 table in a given data range.

Parameters

- **pudl_engine** (`sqlalchemy.engine.Engine`) – SQLAlchemy connection engine for the PUDL DB.
- **pt** (`ImmutableDict`) – a sqlalchemy metadata dictionary of pudl tables
- **freq** (`str`) – a pandas timeseries offset alias. The original data is reported monthly, so the best time frequencies to use here are probably month start (freq='MS') and year start (freq='YS').
- **start_date** (`date-like`) – date-like object, including a string of the form 'YYYY-MM-DD' which will be used to specify the date range of records to be pulled. Dates are inclusive.
- **end_date** (`date-like`) – date-like object, including a string of the form 'YYYY-MM-DD' which will be used to specify the date range of records to be pulled. Dates are inclusive.

Returns A DataFrame containing all records from the EIA 923 Generation table.

Return type `pandas.DataFrame`

```
pudl.output.eia923.generation_fuel_eia923(pudl_engine, pt, freq=None, start_date=None,
                                           end_date=None)
```

Pull records from the generation_fuel_eia923 table in given date range.

Optionally, aggregate the records over some timescale – monthly, yearly, quarterly, etc. as well as by fuel type within a plant.

If the records are not being aggregated, all of the database fields are available. If they're being aggregated, then we preserve the following fields. Per-unit values are re-calculated based on the aggregated totals. Totals are summed across whatever time range is being used, within a given plant and fuel type.

```
* plant_id_eia * report_date * fuel_type_code_pudl * fuel_consumed_units
* fuel_consumed_for_electricity_units * fuel_mmbtu_per_unit *
fuel_consumed_mmbtu * fuel_consumed_for_electricity_mmbtu *
net_generation_mwh
```

In addition, plant and utility names and IDs are pulled in from the EIA 860 tables.

Parameters

- **pudl_engine** (`sqlalchemy.engine.Engine`) – SQLAlchemy connection engine for the PUDL DB.
- **pt** (`ImmutableDict`) – a sqlalchemy metadata dictionary of pudl tables
- **freq** (`str`) – a pandas timeseries offset alias. The original data is reported monthly, so the best time frequencies to use here are probably month start (freq='MS') and year start (freq='YS').
- **start_date** (`date-like`) – date-like object, including a string of the form 'YYYY-MM-DD' which will be used to specify the date range of records to be pulled. Dates are inclusive.
- **end_date** (`date-like`) – date-like object, including a string of the form 'YYYY-MM-DD' which will be used to specify the date range of records to be pulled. Dates are inclusive.

Returns A DataFrame containing all records from the EIA 923 Generation Fuel table.

Return type `pandas.DataFrame`

pudl.output.export module

Routines for exporting data from PUDL for use elsewhere.

Function names should be indicative of the format of the thing that's being exported (e.g. CSV, Excel spreadsheets, parquet files, HDF5).

```
pudl.output.export.annotated_xlsx(df, notes_dict, tags_dict, first_cols, sheet_name,
                                   xlsx_writer)
```

Outputs an annotated spreadsheet workbook based on compiled dataframes.

Creates annotation tab and header rows for EIA 860, EIA 923, and FERC 1 fields in a dataframe. This is done using an Excel Writer object, which must be created and saved outside the function, thereby allowing multiple sheets and associated annotations to be compiled in the same Excel file.

Parameters

- **df** (`pandas.DataFrame`) – The dataframe for which annotations are being created
- **notes_dict** (`dict`) – dictionary with column names as keys and long annotations as values
- **tags_dict** (`dict`) – dictionary of dictionaries with tag categories as keys for outer dictionary and values are dictionaries with column names as keys and values are tag within the tag category
- **first_cols** (`list`) – ordered list of columns that should come first in outfile
- **sheet_name** (`string`) – name of data sheet in output spreadsheet
- **xlsx_writer** (`pandas.ExcelWriter`) – this is an ExcelWriter object used to accumulate multiple tabs, which must be created outside of function, before calling the first time e.g. “xlsx_writer = pd.ExcelWriter('outfile.xlsx')”

Returns which must be called outside the function, after final use of function, for writing out to excel: “xlsx_writer.save()”

Return type `xlsx_writer` (`pandas.ExcelWriter`)

pudl.output.ferc1 module

Functions for pulling FERC Form 1 data out of the PUDL DB.

```
pudl.output.ferc1.fuel_by_plant_ferc1(pudl_engine, pt, thresh=0.5)
```

Summarize FERC fuel data by plant for output.

This is mostly a wrapper around `pudl.transform.ferc1.fuel_by_plant_ferc1` which calculates some summary values on a per-plant basis (as indicated by `utility_id_ferc1` and `plant_name`) related to fuel consumption.

Parameters

- **pudl_engine** (`sqlalchemy.engine.Engine`) – Engine for connecting to the PUDL database.
- **pt** (`immutabledict`) – a sqlalchemy metadata dictionary of pudl tables
- **thresh** (`float`) – Minimum fraction of fuel (cost and mmbtu) required in order for a plant to be assigned a primary fuel. Must be between 0.5 and 1.0. default value is 0.5.

Returns A DataFrame with fuel use summarized by plant.

Return type `pandas.DataFrame`

`pudl.output.ferc1.fuel_ferc1(pudl_engine, pt)`

Pull a useful dataframe related to FERC Form 1 fuel information.

This function pulls the FERC Form 1 fuel data, and joins in the name of the reporting utility, as well as the PUDL IDs for that utility and the plant, allowing integration with other PUDL tables.

Also calculates the total heat content consumed for each fuel, and the total cost for each fuel. Total cost is calculated in two different ways, on the basis of fuel units consumed(e.g. tons of coal, mcf of gas) and on the basis of heat content consumed. In theory these should give the same value for total cost, but this is not always the case.

Todo: Check whether this includes all of the `fuel_ferc1` fields...

Parameters

- **pudl_engine** (`sqlalchemy.engine.Engine`) – Engine for connecting to the PUDL database.
- **pt** (`ImmutableDict`) – a sqlalchemy metadata dictionary of pudl tables

Returns A DataFrame containing useful FERC Form 1 fuel information.

Return type `pandas.DataFrame`

`pudl.output.ferc1.plants_steam_ferc1(pudl_engine, pt)`

Select and joins some useful fields from the FERC Form 1 steam table.

Select the FERC Form 1 steam plant table entries, add in the reporting utility's name, and the PUDL ID for the plant and utility for readability and integration with other tables that have PUDL IDs.

Parameters

- **pudl_engine** (`sqlalchemy.engine.Engine`) – Engine for connecting to the PUDL database.
- **pt** (`ImmutableDict`) – a sqlalchemy metadata dictionary of pudl tables

Returns A DataFrame containing useful fields from the FERC Form 1 steam table.

Return type `pandas.DataFrame`

`pudl.output.ferc1.plants_utils_ferc1(pudl_engine, pt)`

Build a dataframe of useful FERC Plant & Utility information.

Parameters

- **pudl_engine** (`sqlalchemy.engine.Engine`) – Engine for connecting to the PUDL database.
- **pt** (`ImmutableDict`) – a sqlalchemy metadata dictionary of pudl tables

Returns A DataFrame containing useful FERC Form 1 Plant and Utility information.

Return type `pandas.DataFrame`

pudl.output.glue module

Functions that pull glue tables from the PUDL DB for output.

The glue tables hold information that relates our different datasets to each other, for example mapping the FERC plants to EIA generators, or the EIA boilers to EIA generators, or EPA smokestacks to EIA generators.

`pudl.output.glue.boiler_generator_assn` (*pudl_engine*, *pt*, *start_date=None*, *end_date=None*)
Pulls the more complete PUDL/EIA boiler generator associations.

Parameters

- **pudl_engine** (*sqlalchemy.engine.Engine*) – SQLAlchemy connection engine for the PUDL DB.
- **pt** (*ImmutableDict*) – a sqlalchemy metadata dictionary of pudl tables
- **start_date** (*date*) – Date to begin retrieving data.
- **end_date** (*date*) – Date to end retrieving data.

Returns A DataFrame containing the more complete PUDL/EIA boiler generator associations.

Return type `pandas.DataFrame`

pudl.output.pudltabl module

This module provides a class enabling tabular compilations from the PUDL DB.

Many of our potential users are comfortable using spreadsheets, not databases, so we are creating a collection of tabular outputs that contain the most useful core information from the PUDL data packages, including additional keys and human readable names for the objects (utilities, plants, generators) being described in the table.

These tabular outputs can be joined with each other using those keys, and used as a data source within Microsoft Excel, Access, R Studio, or other data analysis packages that folks may be familiar with. They aren't meant to completely replicate all the data and relationships contained within the full PUDL database, but should serve as a generally usable set of PUDL data products.

The PudlTabl class can also provide access to complex derived values, like the generator and plant level marginal cost of electricity (MCOE), which are defined in the analysis module.

In the long run, this is a probably a kind of prototype for pre-packaged API outputs or data products that we might want to be able to provide to users a la carte.

Todo: Return to for update arg and returns values in functions below

```
class pudl.output.pudltabl.PudlTabl (freq=None, start_date=None, end_date=None,  
                                     pudl_engine=None)
```

Bases: `object`

A class for compiling common useful tabular outputs from the PUDL DB.

```
bf_eia923 (update=False)
```

Pull EIA 923 boiler fuel consumption data.

Parameters **update** (*bool*) – If true, re-calculate the output dataframe, even if a cached version exists.

Returns a denormalized table for interactive use.

Return type `pandas.DataFrame`

bga (*update=False*)

Pull the more complete EIA/PUDL boiler-generator associations.

Parameters **update** (*bool*) – If true, re-calculate the output dataframe, even if a cached version exists.

Returns a denormalized table for interactive use.

Return type `pandas.DataFrame`

bga_eia860 (*update=False*)

Pull a dataframe of boiler-generator associations from EIA 860.

Parameters **update** (*bool*) – If true, re-calculate the output dataframe, even if a cached version exists.

Returns a denormalized table for interactive use.

Return type `pandas.DataFrame`

capacity_factor (*update=False, min_cap_fact=None, max_cap_fact=None*)

Calculate and return generator level capacity factors.

Parameters **update** (*bool*) – If true, re-calculate the output dataframe, even if a cached version exists.

Returns a denormalized table for interactive use.

Return type `pandas.DataFrame`

fbp_ferc1 (*update=False*)

Summarize FERC Form 1 fuel usage by plant.

Parameters **update** (*bool*) – If true, re-calculate the output dataframe, even if a cached version exists.

Returns a denormalized table for interactive use.

Return type `pandas.DataFrame`

frc_eia923 (*update=False*)

Pull EIA 923 fuel receipts and costs data.

Parameters **update** (*bool*) – If true, re-calculate the output dataframe, even if a cached version exists.

Returns a denormalized table for interactive use.

Return type `pandas.DataFrame`

fuel_cost (*update=False*)

Calculate and return generator level fuel costs per MWh.

Parameters **update** (*bool*) – If true, re-calculate the output dataframe, even if a cached version exists.

Returns a denormalized table for interactive use.

Return type `pandas.DataFrame`

fuel_ferc1 (*update=False*)

Pull the FERC Form 1 steam plants fuel consumption data.

Parameters **update** (*bool*) – If true, re-calculate the output dataframe, even if a cached version exists.

Returns a denormalized table for interactive use.

Return type `pandas.DataFrame`

gen_eia923 (*update=False*)

Pull EIA 923 net generation data by generator.

Parameters **update** (*bool*) – If true, re-calculate the output dataframe, even if a cached version exists.

Returns a denormalized table for interactive use.

Return type `pandas.DataFrame`

gens_eia860 (*update=False*)

Pull a dataframe describing generators, as reported in EIA 860.

Parameters **update** (*bool*) – If true, re-calculate the output dataframe, even if a cached version exists.

Returns a denormalized table for interactive use.

Return type `pandas.DataFrame`

gf_eia923 (*update=False*)

Pull EIA 923 generation and fuel consumption data.

Parameters **update** (*bool*) – If true, re-calculate the output dataframe, even if a cached version exists.

Returns a denormalized table for interactive use.

Return type `pandas.DataFrame`

hr_by_gen (*update=False*)

Calculate and return generator level heat rates (mmBTU/MWh).

Parameters **update** (*bool*) – If true, re-calculate the output dataframe, even if a cached version exists.

Returns a denormalized table for interactive use.

Return type `pandas.DataFrame`

hr_by_unit (*update=False*)

Calculate and return generation unit level heat rates.

Parameters **update** (*bool*) – If true, re-calculate the output dataframe, even if a cached version exists.

Returns a denormalized table for interactive use.

Return type `pandas.DataFrame`

mcoe (*update=False, min_heat_rate=5.5, min_fuel_cost_per_mwh=0.0, min_cap_fact=0.0, max_cap_fact=1.5*)

Calculate and return generator level MCOE based on EIA data.

Eventually this calculation will include non-fuel operating expenses as reported in FERC Form 1, but for now only the fuel costs reported to EIA are included. They are attributed based on the unit-level heat rates and fuel costs.

Parameters

- **update** (*bool*) – If true, re-calculate the output dataframe, even if a cached version exists.
- **min_heat_rate** – lowest plausible heat rate, in mmBTU/MWh. Any MCOE records with lower heat rates are presumed to be invalid, and are discarded before returning.

- **min_cap_fact** – minimum generator capacity factor. Generator records with a lower capacity factor will be filtered out before returning. This allows the user to exclude generators that aren't being used enough to have valid.
- **min_fuel_cost_per_mwh** – minimum fuel cost on a per MWh basis that is required for a generator record to be considered valid. For some reason there are now a large number of \$0 fuel cost records, which previously would have been NaN.
- **max_cap_fact** – maximum generator capacity factor. Generator records with a lower capacity factor will be filtered out before returning. This allows the user to exclude generators that aren't being used enough to have valid.

Returns a compilation of generator attributes, including fuel costs per MWh.

Return type `pandas.DataFrame`

own_eia860 (*update=False*)

Pull a dataframe of generator level ownership data from EIA 860.

Parameters **update** (*bool*) – If true, re-calculate the output dataframe, even if a cached version exists.

Returns a denormalized table for interactive use.

Return type `pandas.DataFrame`

plants_eia860 (*update=False*)

Pull a dataframe of plant level info reported in EIA 860.

Parameters **update** (*bool*) – If true, re-calculate the output dataframe, even if a cached version exists.

Returns a denormalized table for interactive use.

Return type `pandas.DataFrame`

plants_steam_ferc1 (*update=False*)

Pull the FERC Form 1 steam plants data.

Parameters **update** (*bool*) – If true, re-calculate the output dataframe, even if a cached version exists.

Returns a denormalized table for interactive use.

Return type `pandas.DataFrame`

pu_eia860 (*update=False*)

Pull a dataframe of EIA plant-utility associations.

Parameters **update** (*bool*) – If true, re-calculate the output dataframe, even if a cached version exists.

Returns a denormalized table for interactive use.

Return type `pandas.DataFrame`

pu_ferc1 (*update=False*)

Pull a dataframe of FERC plant-utility associations.

Parameters **update** (*bool*) – If true, re-calculate the output dataframe, even if a cached version exists.

Returns a denormalized table for interactive use.

Return type `pandas.DataFrame`

utils_eia860 (*update=False*)

Pull a dataframe describing utilities reported in EIA 860.

Parameters **update** (*bool*) – If true, re-calculate the output dataframe, even if a cached version exists.

Returns a denormalized table for interactive use.

Return type `pandas.DataFrame`

`pudl.output.pudltabl.get_table_meta` (*pudl_engine*)

Grab the pudl sqlite database table metadata.

Module contents

Useful post-processing and denormalized outputs based on PUDL.

The datapackages which are output by the PUDL ETL pipeline are well normalized and suitable for use as relational database tables. This minimizes data duplication and helps avoid many kinds of data corruption and the potential for internal inconsistency. However, that's not always the easiest kind of data to work with. Sometimes we want all the names and IDs in a single dataframe or table, for human readability. Sometimes you want the useful derived values.

This subpackage compiles a bunch of outputs we found we were commonly generating, so that they can be done automatically and uniformly. They are encapsulated within the `pudl.output.pudltabl.PudlTabl` class.

pudl.transform package

Submodules

pudl.transform.eia module

Routines specific to cleaning up EIA data.

This module helps with the normalization of EIA datasets and complinging additional connections between EIA entities. Right now, these two tasks include what we call harvesting and generating a more complete set of boiler generator associations. The harvesting process normalizes the EIA tables - it consolidates the duplicated fields/records into entity and annual entity tables. The boiler generator associations (bga) takes the given 860 bga and expands on this through several methods within the `_boiler_generator_assn` function.

```
pudl.transform.eia.transform(eia_transformed_dfs, eia923_years=(2009, 2010, 2011, 2012,
2013, 2014, 2015, 2016, 2017), eia860_years=(2011, 2012, 2013,
2014, 2015, 2016, 2017), debug=False)
```

Creates DataFrames for EIA Entity tables and modifies EIA tables.

This function coordinates two main actions: generating the entity tables via `_harvesting()` and generating the boiler generator associations via `_boiler_generator_assn()`.

There is also some removal of tables that are no longer needed after the entity harvesting is finished.

Parameters

- **eia_transformed_dfs** (*dict*) – a dictionary of table names (keys) and transformed dataframes (values).
- **eia923_years** (*list*) – a list of years for EIA 923
- **eia860_years** (*list*) – a list of years for EIA 860
- **debug** (*bool*) – if true, informational columns will be added into boiler_generator_assn

Returns

a dictionary of table names (keys) and dataframes (values) for the entity tables.

`eia_transformed_dfs` (*dict*): a dictionary of table names (keys) and dataframes (values) for the rest of the EIA tables.

Return type `entities_dfs` (*dict*)

pudl.transform.eia860 module

Module to perform data cleaning functions on EIA860 data tables.

`pudl.transform.eia860.boiler_generator_assn` (*eia860_dfs, eia860_transformed_dfs*)

Pulls and transforms the boiler generator association table.

Parameters

- **`eia860_dfs`** (*dict*) – Each entry in this dictionary of DataFrame objects corresponds to a page from the EIA860 form, as reported in the Excel spreadsheets they distribute.
- **`eia860_transformed_dfs`** (*dict*) – A dictionary of DataFrame objects in which pages from EIA860 form (keys) correspond to normalized DataFrames of values from that page (values)

Returns `eia860_transformed_dfs`, a dictionary of DataFrame objects in which pages from EIA860 form (keys) correspond to normalized DataFrames of values from that page (values)

Return type *dict*

`pudl.transform.eia860.generators` (*eia860_dfs, eia860_transformed_dfs*)

Pulls and transforms the generators table.

There are three tabs that the generator records come from (proposed, existing, and retired). We pull each tab into one dataframe and include an 'operational_status' to indicate which tab the record came from.

Parameters

- **`eia860_dfs`** (*dict*) – Each entry in this dictionary of DataFrame objects corresponds to a page from the EIA860 form, as reported in the Excel spreadsheets they distribute.
- **`eia860_transformed_dfs`** (*dict*) – A dictionary of DataFrame objects in which pages from EIA860 form (keys) correspond to a normalized DataFrame of values from that page (values)

Returns `eia860_transformed_dfs`, a dictionary of DataFrame objects in which pages from EIA860 form (keys) correspond to normalized DataFrames of values from that page (values)

Return type *dict*

`pudl.transform.eia860.ownership` (*eia860_dfs, eia860_transformed_dfs*)

Pulls and transforms the ownership table.

Parameters

- **`eia860_dfs`** (*dict*) – Each entry in this dictionary of DataFrame objects corresponds to a page from the EIA860 form, as reported in the Excel spreadsheets they distribute
- **`eia860_transformed_dfs`** (*dict*) – A dictionary of DataFrame objects in which pages from EIA860 form (keys) correspond to normalized DataFrames of values from that page (values)

Returns `eia860_transformed_dfs`, a dictionary of DataFrame objects in which pages from EIA860 form (keys) correspond to normalized DataFrames of values from that page (values)

Return type `dict`

Todo: Convert assert statement to AssertionError

`pudl.transform.eia860.plants` (`eia860_dfs`, `eia860_transformed_dfs`)

Pulls and transforms the plants table.

Much of the static plant information is reported repeatedly, and scattered across several different pages of EIA 923. The data frame which this function uses is assembled from those many different pages, and passed in via the same dictionary of dataframes that all the other ingest functions use for uniformity.

Parameters

- **`eia860_dfs`** (`dict`) – Each entry in this dictionary of DataFrame objects corresponds to a page from the EIA860 form, as reported in the Excel spreadsheets they distribute.
- **`eia860_transformed_dfs`** (`dict`) – A dictionary of DataFrame objects in which pages from EIA860 form (keys) correspond to normalized DataFrames of values from that page (values)

Returns `eia860_transformed_dfs`, a dictionary of DataFrame objects in which pages from EIA860 form (keys) correspond to normalized DataFrames of values from that page (values)

Return type `dict`

`pudl.transform.eia860.transform` (`eia860_raw_dfs`, `eia860_tables=``('boiler_generator_assn_eia860', 'utilities_eia860', 'plants_eia860', 'generators_eia860', 'ownership_eia860')`)

Transforms EIA 860 DataFrames.

Parameters

- **`eia860_raw_dfs`** (`dict`) – a dictionary of tab names (keys) and DataFrames (values). This can be generated by `pudl`.
- **`eia860_tables`** (`tuple`) – A tuple containing the names of the EIA 860 tables that can be pulled into PUDL

Returns A dictionary of DataFrame objects in which pages from EIA860 form (keys) corresponds to a normalized DataFrame of values from that page (values)

Return type `dict`

`pudl.transform.eia860.utilities` (`eia860_dfs`, `eia860_transformed_dfs`)

Pulls and transforms the utilities table.

Parameters

- **`eia860_dfs`** (`dict`) – Each entry in this dictionary of DataFrame objects corresponds to a page from the EIA860 form, as reported in the Excel spreadsheets they distribute.
- **`eia860_transformed_dfs`** (`dict`) – A dictionary of DataFrame objects in which pages from EIA860 form (keys) correspond to normalized DataFrames of values from that page (values)

Returns `eia860_transformed_dfs`, a dictionary of DataFrame objects in which pages from EIA860 form (keys) correspond to normalized DataFrames of values from that page (values)

Return type `dict`

pudl.transform.eia923 module

Routines specific to cleaning up EIA Form 923 data.

`pudl.transform.eia923.boiler_fuel(eia923_dfs, eia923_transformed_dfs)`

Transforms the boiler_fuel_eia923 table.

Parameters

- **eia923_dfs** (*dict*) – Each entry in this dictionary of DataFrame objects corresponds to a page from the EIA923 form, as reported in the Excel spreadsheets they distribute.
- **eia923_transformed_dfs** (*dict*) – A dictionary of DataFrame objects in which pages from EIA923 form (keys) correspond to normalized DataFrames of values from that page (values)

Returns eia923_transformed_dfs, a dictionary of DataFrame objects in which pages from EIA923 form (keys) correspond to normalized DataFrames of values from that page (values).

Return type dict

`pudl.transform.eia923.coalmine(eia923_dfs, eia923_transformed_dfs)`

Transforms the coalmine_eia923 table.

Parameters

- **eia923_dfs** (*dict*) – Each entry in this dictionary of DataFrame objects corresponds to a page from the EIA923 form, as reported in the Excel spreadsheets they distribute.
- **eia923_transformed_dfs** (*dict*) – A dictionary of DataFrame objects in which pages from EIA923 form (keys) correspond to normalized DataFrames of values from that page (values)

Returns eia923_transformed_dfs, a dictionary of DataFrame objects in which pages from EIA923 form (keys) correspond to normalized DataFrames of values from that page (values).

Return type dict

`pudl.transform.eia923.fuel_receipts_costs(eia923_dfs, eia923_transformed_dfs)`

Transforms the fuel_receipts_costs_eia923 dataframe.

Fuel cost is reported in cents per mmbtu. Converts cents to dollars.

Parameters

- **eia923_dfs** (*dict*) – Each entry in this dictionary of DataFrame objects corresponds to a page from the EIA923 form, as reported in the Excel spreadsheets they distribute.
- **eia923_transformed_dfs** (*dict*) – A dictionary of DataFrame objects in which pages from EIA923 form (keys) correspond to normalized DataFrames of values from that page (values)

Returns eia923_transformed_dfs, a dictionary of DataFrame objects in which pages from EIA923 form (keys) correspond to normalized DataFrames of values from that page (values)

Return type dict

`pudl.transform.eia923.generation(eia923_dfs, eia923_transformed_dfs)`

Transforms the generation_eia923 table.

Parameters

- **eia923_dfs** (*dict*) – Each entry in this dictionary of DataFrame objects corresponds to a page from the EIA923 form, as reported in the Excel spreadsheets they distribute.

- **eia923_transformed_dfs** (*dict*) – A dictionary of DataFrame objects in which pages from EIA923 form (keys) correspond to normalized DataFrames of values from that page (values)

Returns eia923_transformed_dfs, a dictionary of DataFrame objects in which pages from EIA923 form (keys) correspond to normalized DataFrames of values from that page (values).

Return type *dict*

`pudl.transform.eia923.generation_fuel` (*eia923_dfs, eia923_transformed_dfs*)
Transforms the generation_fuel_eia923 table.

Parameters

- **eia923_dfs** (*dict*) – Each entry in this dictionary of DataFrame objects corresponds to a page from the EIA923 form, as reported in the Excel spreadsheets they distribute.
- **eia923_transformed_dfs** (*dict*) – A dictionary of DataFrame objects in which pages from EIA923 form (keys) correspond to normalized DataFrames of values from that page (values)

Returns eia923_transformed_dfs, a dictionary of DataFrame objects in which pages from EIA923 form (keys) correspond to normalized DataFrames of values from that page (values).

Return type *dict*

`pudl.transform.eia923.plants` (*eia923_dfs, eia923_transformed_dfs*)
Transforms the plants_eia923 table.

Much of the static plant information is reported repeatedly, and scattered across several different pages of EIA 923. The data frame that this function uses is assembled from those many different pages, and passed in via the same dictionary of dataframes that all the other ingest functions use for uniformity.

Parameters

- **eia923_dfs** (*dictionary of pandas.DataFrame*) – Each entry in this dictionary of DataFrame objects corresponds to a page from the EIA 923 form, as reported in the Excel spreadsheets they distribute.
- **eia923_transformed_dfs** (*dict*) – A dictionary of DataFrame objects in which pages from EIA923 form (keys) correspond to normalized DataFrames of values from that page (values)

Returns eia923_transformed_dfs, a dictionary of DataFrame objects in which pages from EIA923 form (keys) correspond to normalized DataFrames of values from that page (values)

Return type *dict*

`pudl.transform.eia923.transform` (*eia923_raw_dfs, eia923_tables=('generation_fuel_eia923', 'boiler_fuel_eia923', 'generation_eia923', 'coalmine_eia923', 'fuel_receipts_costs_eia923')*)

Transforms all the EIA 923 tables.

Parameters

- **eia923_raw_dfs** (*dict*) – a dictionary of tab names (keys) and DataFrames (values). Generated from `pudl.extract.eia923.extract()`.
- **eia923_tables** (*tuple*) – A tuple containing the EIA923 tables that can be pulled into PUDL.

Returns A dictionary of DataFrame objects in which pages from EIA923 form (keys) corresponds to a normalized DataFrame of values from that page (values)

Return type *dict*

pudl.transform.epacems module

Routines specific to cleaning up EPA CEMS hourly data.

`pudl.transform.epacems.add_facility_id_unit_id_epa(df)`

Harmonize columns that are added later.

The load into Postgres checks for consistent column names, and these two columns aren't present before August 2008, so this adds them in.

Parameters `df` (`pd.DataFrame`) – A CEMS dataframe

Returns The same DataFrame guaranteed to have int `facility_id` and `unit_id_epa` cols

`pudl.transform.epacems.correct_gross_load_mw(df)`

Fix values of gross load that are wrong by orders of magnitude.

Parameters `df` (`pd.DataFrame`) – A CEMS dataframe

Returns The same DataFrame with corrected gross load values.

Return type `pd.DataFrame`

`pudl.transform.epacems.fix_up_dates(df, plant_utc_offset)`

Fix the dates for the CEMS data.

Parameters

- `df` (`pandas.DataFrame`) – A CEMS hourly dataframe for one year-month-state
- `plant_utc_offset` (`pandas.DataFrame`) – A dataframe of plants' timezones

Returns The same data, with an `op_datetime_utc` column added and the `op_date` and `op_hour` columns removed

Return type `pandas.DataFrame`

`pudl.transform.epacems.harmonize_eia_epa_orispl(df)`

Harmonize the ORISPL code to match the EIA data – NOT YET IMPLEMENTED.

The EIA plant IDs and CEMS ORISPL codes almost match, but not quite. See https://www.epa.gov/sites/production/files/2018-02/documents/egrid2016_technicalsupportdocument_0.pdf#page=104 for an example.

Note that this transformation needs to be run *before* `fix_up_dates`, because `fix_up_dates` uses the plant ID to look up timezones.

Parameters `df` (`pandas.DataFrame`) – A CEMS hourly dataframe for one year-month-state

Returns The same data, with the ORISPL plant codes corrected to match the EIA plant IDs.

Return type `pandas.DataFrame`

Todo: Ctually implement the function...

`pudl.transform.epacems.transform(epacems_raw_dfs, pkg_dir)`

Transform EPA CEMS hourly data for use in datapackage export.

To Do: Incomplete docstring.

pudl.transform.epaipm module

Module to perform data cleaning functions on EPA IPM data tables.

`pudl.transform.epaipm.load_curves` (*epaipm_dfs*, *epaipm_transformed_dfs*)

Transform the load curve table from wide to tidy format.

Parameters

- **epaipm_dfs** (*dict*) – Each entry in this dictionary of DataFrame objects corresponds to a table from EPA’s IPM, as reported in the Excel spreadsheets they distribute.
- **epa_epaipm_transformed_dfs** (*dict*) – A dictionary of DataFrame objects in which tables from EPA IPM (keys) correspond to normalized DataFrames of values from that table (values)

Returns A dictionary of DataFrame objects in which tables from EPA IPM (keys) correspond to normalized DataFrames of values from that table (values)

Return type *dict*

`pudl.transform.epaipm.plant_region_map` (*epaipm_dfs*, *epaipm_transformed_dfs*)

Transforms the map of plant ids to IPM regions for all plants.

Parameters

- **epaipm_dfs** (*dict*) – Each entry in this dictionary of DataFrame objects corresponds to a table from EPA’s IPM, as reported in the Excel spreadsheets they distribute.
- **epaipm_transformed_dfs** (*dict*) – A dictionary of DataFrame objects in which tables from EPA IPM(keys) correspond to normalized DataFrames of values from that table(values)

Returns A dictionary of DataFrame objects in which tables from EPA IPM(keys) correspond to normalized DataFrames of values from that table(values)

Return type *dict*

`pudl.transform.epaipm.transform` (*epaipm_raw_dfs*, *epaipm_tables*=('transmission_single_epaipm',
'transmission_joint_epaipm', 'load_curves_epaipm',
'plant_region_map_epaipm'))

Transform EPA IPM DataFrames.

Parameters

- **epaipm_raw_dfs** (*dict*) – a dictionary of table names(keys) and DataFrames(values)
- **epaipm_tables** (*list*) – The list of EPA IPM tables that can be successfully pulled into PUDL

Returns A dictionary of DataFrame objects in which tables from EPA IPM(keys) correspond to normalized DataFrames of values from that table(values)

Return type *dict*

`pudl.transform.epaipm.transmission_joint` (*epaipm_dfs*, *epaipm_transformed_dfs*)

Transforms transmission constraints between multiple inter-regional links.

Parameters

- **epaipm_dfs** (*dict*) – Each entry in this dictionary of DataFrame objects corresponds to a table from EPA’s IPM, as reported in the Excel spreadsheets they distribute.

- **epa_epaipm_transformed_dfs** (*dict*) – A dictionary of DataFrame objects in which tables from EPA IPM (keys) correspond to normalized DataFrames of values from that table (values)

Returns A dictionary of DataFrame objects in which tables from EPA IPM (keys) correspond to normalized DataFrames of values from that table (values)

Return type `dict`

`pudl.transform.epaipm.transmission_single(epaipm_dfs, epaipm_transformed_dfs)`

Transforms the transmission constraints between individual regions.

Parameters

- **epaipm_dfs** (*dict*) – Each entry in this dictionary of DataFrame objects corresponds to a table from EPA’s IPM, as reported in the Excel spreadsheets they distribute.
- **epa_epaipm_transformed_dfs** (*dict*) – A dictionary of DataFrame objects in which tables from EPA IPM (keys) correspond to normalized DataFrames of values from that table (values)

Returns A dictionary of DataFrame objects in which tables from EPA IPM (keys) correspond to normalized DataFrames of values from that table (values)

Return type `dict`

pudl.transform.ferc1 module

Routines for transforming FERC Form 1 data before loading into the PUDL DB.

This module provides a variety of functions that are used in cleaning up the FERC Form 1 data prior to loading into our database. This includes adopting standardized units and column names, standardizing the formatting of some string values, and correcting data entry errors which we can infer based on the existing data. It may also include removing bad data, or replacing it with the appropriate NA values.

class `pudl.transform.ferc1.FERCPlantClassifier` (*min_sim=0.75, plants_df=None*)

Bases: `sklearn.base.BaseEstimator`, `sklearn.base.ClassifierMixin`

A classifier for identifying FERC plant time series in FERC Form 1 data.

We want to be able to give the classifier a FERC plant record, and get back the group of records(or the ID of the group of records) that it ought to be part of.

There are hundreds of different groups of records, and we can only know what they are by looking at the whole dataset ahead of time. This is the “fitting” step, in which the groups of records resulting from a particular set of model parameters(e.g. the weights that are attributes of the class) are generated.

Once we have that set of record categories, we can test how well the classifier performs, by checking it against test / training data which we have already classified by hand. The test / training set is a list of lists of unique FERC plant record IDs(each record ID is the concatenation of: report year, respondent id, supplement number, and row number). It could also be stored as a dataframe where each column is associated with a year of data(some of which could be empty). Not sure what the best structure would be.

If it’s useful, we can assign each group a unique ID that is the time ordered concatenation of each of the constituent record IDs. Need to understand what the process for checking the classification of an input record looks like.

To score a given classifier, we can look at what proportion of the records in the test dataset are assigned to the same group as in our manual classification of those records. There are much more complicated ways to do the scoring too... but for now let’s just keep it as simple as possible.

fit (*X*, *y=None*)

Use weighted FERC plant features to group records into time series.

The fit method takes the vectorized, normalized, weighted FERC plant features (*X*) as input, calculates the pairwise cosine similarity matrix between all records, and groups the records in their best time series. The similarity matrix and best time series are stored as data members in the object for later use in scoring & predicting.

This isn't quite the way a fit method would normally work.

Parameters

- (*y*) – a sparse matrix of size *n_samples* x *n_features*.
- () –

Returns

Return type `pandas.DataFrame`

Todo: Zane revisit args and returns

predict (*X*, *y=None*)

Identify time series of similar records to input *record_ids*.

Given a one-dimensional dataframe *X*, containing FERC record IDs, return a dataframe in which each row corresponds to one of the input *record_id* values (ordered as the input was ordered), with each column corresponding to one of the years worth of data. Values in the returned dataframe are the FERC *record_ids* of the record most similar to the input record within that year. Some of them may be null, if there was no sufficiently good match.

Row index is the seed record IDs. Column index is years.

TODO: * This method is hideously inefficient. It should be vectorized. * There's a line that throws a FutureWarning that needs to be fixed.

score (*X*, *y=None*)

Scores a collection of FERC plant categorizations.

For every record ID in *X*, predict its record group and calculate a metric of similarity between the prediction and the "ground truth" group that was passed in for that value of *X*.

Parameters

- **X** (`pandas.DataFrame`) – an *n_samples* x 1 pandas dataframe of FERC Form 1 record IDs.
- **y** (`pandas.DataFrame`) – a dataframe of "ground truth" FERC Form 1 record groups, corresponding to the list record IDs in *X*

Returns The average of all the similarity metrics as the score.

Return type `numpy.ndarray`

transform (*X*, *y=None*)

Passthrough transform method – just returns self.

`pudl.transform.ferc1.accumulated_depreciation` (*ferc1_raw_dfs*, *ferc1_transformed_dfs*)

Transforms FERC Form 1 depreciation data for loading into PUDL.

This information is organized by FERC account, with each line of the FERC Form 1 having a different descriptive identifier like 'balance_end_of_year' or 'transmission'.

Parameters

- **ferc1_raw_dfs** (*dict*) – Each entry in this dictionary of DataFrame objects corresponds to a table from the FERC Form 1 DBC database.
- **ferc1_transformed_dfs** (*dict*) – A dictionary of DataFrames to be transformed.

Returns The dictionary of the transformed DataFrames.

Return type *dict*

```
pudl.transform.ferc1.fuel(ferc1_raw_dfs, ferc1_transformed_dfs)
```

Transforms FERC Form 1 fuel data for loading into PUDL Database.

This process includes converting some columns to be in terms of our preferred units, like MWh and mmbtu instead of kWh and btu. Plant names are also standardized (stripped & Title Case). Fuel and fuel unit strings are also standardized using our `cleanstrings()` function and string cleaning dictionaries found in `pudl.constants`.

Parameters

- **ferc1_raw_dfs** (*dict*) – Each entry in this dictionary of DataFrame objects corresponds to a table from the FERC Form 1 DBC database.
- **ferc1_transformed_dfs** (*dict*) – A dictionary of DataFrames to be transformed.

Returns The dictionary of transformed dataframes.

Return type *dict*

```
pudl.transform.ferc1.fuel_by_plant_ferc1(fuel_df, thresh=0.5)
```

Calculates useful FERC Form 1 fuel metrics on a per plant-year basis.

Each record in the FERC Form 1 corresponds to a particular type of fuel. Many plants – especially coal plants – use more than one fuel, with gas and/or diesel serving as startup fuels. In order to be able to classify the type of plant based on relative proportions of fuel consumed or fuel costs it is useful to aggregate these per-fuel records into a single record for each plant.

Fuel cost (in nominal dollars) and fuel heat content (in mmBTU) are calculated for each fuel based on the cost and heat content per unit, and the number of units consumed, and then summed by fuel type (there can be more than one record for a given type of fuel in each plant because we are simplifying the fuel categories). The per-fuel records are then pivoted to create one column per fuel type. The total is summed and stored separately, and the individual fuel costs & heat contents are divided by that total, to yield fuel proportions. Based on those proportions and a minimum threshold that’s passed in, a “primary” fuel type is then assigned to the plant-year record and given a string label.

Parameters

- **fuel_df** (*pandas.DataFrame*) – Pandas DataFrame resembling the post-transform result for the `fuel_ferc1` table.
- **thresh** (*float*) – A value between 0.5 and 1.0 indicating the minimum fraction of overall heat content that must have been provided by a fuel in a plant-year for it to be considered the “primary” fuel for the plant in that year. Default value: 0.5.

Returns A DataFrame with a single record for each

Return type *pandas.DataFrame*

plant-year, including the columns required to merge it with the `plants_steam_ferc1` table/DataFrame (report_year, utility_id_ferc1, and plant_name) as well as totals for fuel mmbtu consumed in that plant-year, and the cost of fuel in that year, the proportions of heat content and fuel costs for each fuel in that year, and a column that labels the plant’s primary fuel for that year.

Raises `AssertionError` – If the DataFrame input does not have the columns required to run the function.

```
pudl.transform.ferc1.make_ferc_clf(plants_df, ngram_min=2, ngram_max=10, min_sim=0.75,  
                                   plant_name_wt=2.0, plant_type_wt=2.0, construc-  
                                   tion_type_wt=1.0, capacity_mw_wt=1.0, construc-  
                                   tion_year_wt=1.0, utility_id_ferc1_wt=1.0,  
                                   fuel_fraction_wt=1.0)
```

Create a FERC Plant Classifier using several weighted features.

Given a FERC steam plants dataframe `plants_df`, which also includes fuel consumption information, transform a selection of useful columns into features suitable for use in calculating inter-record cosine similarities. Individual features are weighted according to the keyword arguments.

Features include:

- `plant_name` (via TF-IDF, with `ngram_min` and `ngram_max` as parameters)
- `plant_type` (OneHot encoded categorical feature)
- `construction_type` (OneHot encoded categorical feature)
- `capacity_mw` (MinMax scaled numerical feature)
- `construction_year` (OneHot encoded categorical feature)
- `utility_id_ferc1` (OneHot encoded categorical feature)
- `fuel_fraction_mmbtu` (several MinMax scaled numerical columns, which are normalized and treated as a single feature.)

This feature matrix is then used to instantiate a `FERCPlantClassifier`.

The combination of the `ColumnTransformer` and `FERCPlantClassifier` are combined in a `sklearn Pipeline`, which is returned by the function.

Parameters

- **`ngram_min`** (*int*) – the minimum lengths to consider in the vectorization of the `plant_name` feature.
- **`ngram_max`** (*int*) – the maximum n-gram lengths to consider in the vectorization of the `plant_name` feature.
- **`min_sim`** (*float*) – the minimum cosine similarity between two records that can be considered a “match” (a number between 0.0 and 1.0).
- **`plant_name_wt`** (*float*) – weight used to determine the relative importance of each of the features in the feature matrix used to calculate the cosine similarity between records. Used to scale each individual feature before the vectors are normalized.
- **`plant_type_wt`** (*float*) – weight used to determine the relative importance of each of the features in the feature matrix used to calculate the cosine similarity between records. Used to scale each individual feature before the vectors are normalized.
- **`construction_type_wt`** (*float*) – weight used to determine the relative importance of each of the features in the feature matrix used to calculate the cosine similarity between records. Used to scale each individual feature before the vectors are normalized.
- **`capacity_mw_wt`** (*float*) – weight used to determine the relative importance of each of the features in the feature matrix used to calculate the cosine similarity between records. Used to scale each individual feature before the vectors are normalized.

- **construction_year_wt** (*float*) – weight used to determine the relative importance of each of the features in the feature matrix used to calculate the cosine similarity between records. Used to scale each individual feature before the vectors are normalized.
- **utility_id_ferc1_wt** (*float*) – weight used to determine the relative importance of each of the features in the feature matrix used to calculate the cosine similarity between records. Used to scale each individual feature before the vectors are normalized.
- **fuel_fraction_wt** (*float*) – weight used to determine the relative importance of each of the features in the feature matrix used to calculate the cosine similarity between records. Used to scale each individual feature before the vectors are normalized.

Returns an sklearn Pipeline that performs reprocessing and classification with a FERCPlantClassifier object.

Return type `sklearn.pipeline.Pipeline`

`pudl.transform.ferc1.plant_in_service(ferc1_raw_dfs, ferc1_transformed_dfs)`

Transforms FERC Form 1 plant_in_service data for loading into PUDL.

This information is organized by FERC account, with each line of the FERC Form 1 having a different FERC account id (most are numeric and correspond to FERC’s Uniform Electric System of Accounts). As of PUDL v0.1, this data is only valid from 2007 onward, as the line numbers for several accounts are different in earlier years.

Parameters

- **ferc1_raw_dfs** (*dict*) – Each entry in this dictionary of DataFrame objects corresponds to a table from the FERC Form 1 DBC database.
- **ferc1_transformed_dfs** (*dict*) – A dictionary of DataFrames to be transformed.

Returns The dictionary of the transformed DataFrames.

Return type `dict`

`pudl.transform.ferc1.plants_hydro(ferc1_raw_dfs, ferc1_transformed_dfs)`

Transforms FERC Form 1 plant_hydro data for loading into PUDL Database.

Standardizes plant names (stripping whitespace and Using Title Case). Also converts into our preferred units of MW and MWh.

Parameters

- **ferc1_raw_dfs** (*dict*) – Each entry in this dictionary of DataFrame objects corresponds to a table from the FERC Form 1 DBC database.
- **ferc1_transformed_dfs** (*dict*) – A dictionary of DataFrames to be transformed.

Returns The dictionary of transformed dataframes.

Return type `dict`

`pudl.transform.ferc1.plants_pumped_storage(ferc1_raw_dfs, ferc1_transformed_dfs)`

Transforms FERC Form 1 pumped storage data for loading into PUDL.

Standardizes plant names (stripping whitespace and Using Title Case). Also converts into our preferred units of MW and MWh.

Parameters

- **ferc1_raw_dfs** (*dict*) – Each entry in this dictionary of DataFrame objects corresponds to a table from the FERC Form 1 DBC database.
- **ferc1_transformed_dfs** (*dict*) – A dictionary of DataFrames to be transformed.

Returns The dictionary of transformed dataframes.

Return type `dict`

```
pudl.transform.ferc1.plants_small(ferc1_raw_dfs, ferc1_transformed_dfs)
```

Transforms FERC Form 1 plant_small data for loading into PUDL Database.

This FERC Form 1 table contains information about a large number of small plants, including many small hydroelectric and other renewable generation facilities. Unfortunately the data is not well standardized, and so the plants have been categorized manually, with the results of that categorization stored in an Excel spreadsheet. This function reads in the plant type data from the spreadsheet and merges it with the rest of the information from the FERC DB based on record number, FERC respondent ID, and report year. When possible the FERC license number for small hydro plants is also manually extracted from the data.

This categorization will need to be renewed with each additional year of FERC data we pull in. As of v0.1 the small plants have been categorized for 2004-2015.

Parameters

- **ferc1_raw_dfs** (*dict*) – Each entry in this dictionary of DataFrame objects corresponds to a table from the FERC Form 1 DBC database.
- **ferc1_transformed_dfs** (*dict*) – A dictionary of DataFrames to be transformed.

Returns The dictionary of transformed dataframes.

Return type `dict`

```
pudl.transform.ferc1.plants_steam(ferc1_raw_dfs, ferc1_transformed_dfs)
```

Transforms FERC Form 1 plant_steam data for loading into PUDL Database.

This includes converting to our preferred units of MWh and MW, as well as standardizing the strings describing the kind of plant and construction.

Parameters

- **ferc1_raw_dfs** (*dict*) – Each entry in this dictionary of DataFrame objects corresponds to a table from the FERC Form 1 DBC database.
- **ferc1_transformed_dfs** (*dict*) – A dictionary of DataFrames to be transformed.

Returns of transformed dataframes, including the newly transformed plants_steam_ferc1 dataframe.

Return type `dict`

```
pudl.transform.ferc1.plants_steam_validate_ids(ferc1_steam_df)
```

Tests that plant_id_ferc1 times series includes one record per year.

Parameters **ferc1_steam_df** (*pandas.DataFrame*) – A DataFrame of the data from the FERC 1 Steam table.

Returns None

```
pudl.transform.ferc1.purchased_power(ferc1_raw_dfs, ferc1_transformed_dfs)
```

Transforms FERC Form 1 pumped storage data for loading into PUDL.

This table has data about inter-utility power purchases into the PUDL DB. This includes how much electricity was purchased, how much it cost, and who it was purchased from. Unfortunately the field describing which other utility the power was being bought from is poorly standardized, making it difficult to correlate with other data. It will need to be categorized by hand or with some fuzzy matching eventually.

Parameters

- **ferc1_raw_dfs** (*dict*) – Each entry in this dictionary of DataFrame objects corresponds to a table from the FERC Form 1 DBC database.

- **ferc1_transformed_dfs** (*dict*) – A dictionary of DataFrames to be transformed.

Returns The dictionary of the transformed DataFrames.

Return type *dict*

```
pudl.transform.ferc1.transform(ferc1_raw_dfs, ferc1_tables=('fuel_ferc1',
                                                           'plants_small_ferc1',
                                                           'plants_hydro_ferc1',
                                                           'plants_pumped_storage_ferc1',
                                                           'plant_in_service_ferc1',
                                                           'purchased_power_ferc1',
                                                           'accumulated_depreciation_ferc1'))
```

Transforms FERC 1.

Parameters

- **ferc1_raw_dfs** (*dict*) – Each entry in this dictionary of DataFrame objects corresponds to a table from the FERC Form 1 DBC database
- **ferc1_tables** (*tuple*) – A tuple containing the set of tables which have been successfully integrated into PUDL

Returns A dictionary of the transformed DataFrames.

Return type *dict*

Module contents

Modules implementing the “Transform” step of the PUDL ETL pipeline.

Each module in this subpackage transforms the tabular data associated with a single data source from the PUDL *Data Catalog*. This process begins with a dictionary of “raw” pandas.DataFrame’s produced by the corresponding data source specific routines from the :mod:`pudl.extract` subpackage, and ends with a dictionary of pandas.DataFrame’s that are fully normalized, cleaned, and congruent with the tabular datapackage metadata -- i.e. they are ready to be exported by the :mod:`pudl.load` module.

pudl.workspace package

Submodules

pudl.workspace.datastore module

Download the original public data sources used by PUDL.

This module provides programmatic, platform-independent access to the original data sources which are used to populate the PUDL database. Those sources currently include: FERC Form 1, EIA Form 860, and EIA Form 923. The module can be used to download the data, and populate a local data store which is organized such that the rest of the PUDL package knows where to find all the raw data it needs.

Support for selectively downloading portions of the EPA’s large Continuous Emissions Monitoring System dataset will be added in the future.

```
pudl.workspace.datastore.assert_valid_param(source, year, month=None, state=None,
                                             check_month=None)
```

Check whether parameters used in various datastore functions are valid.

Parameters

- **source** (*str*) – A string indicating which data source we are going to be downloading. Currently it must be one of the following: - ‘eia860’ - ‘eia861’ - ‘eia923’ - ‘ferc1’ - ‘epacems’
- **year** (*int or None*) – the year for which data should be downloaded. Must be within the range of valid data years, which is specified for each data source in the pudl.constants module. Use None for data sources that do not have multiple years.
- **month** (*int*) – the month for which data should be downloaded. Only used for EPA CEMS.
- **state** (*str*) – the state for which data should be downloaded. Only used for EPA CEMS.
- **check_month** –

Todo: Return to - what is check_month?

Raises

- **AssertionError** – If the source is not among the list of valid sources.
- **AssertionError** – If the source is not found in the valid data years.
- **AssertionError** – If the year is not valid for the specified source.
- **AssertionError** – If the source is not found in valid base download URLs.
- **AssertionError** – If the month is not valid (1-12).
- **AssertionError** – If the state is not a valid US state abbreviation.

`pudl.workspace.datastore.check_if_need_update(source, year, states, data_dir, clobber=False)`

Check to see if the file is already downloaded and clobber is False.

Do we really need to download the requested data? Only case in which we don’t have to do anything is when the downloaded file already exists and clobber is False.

Parameters

- **source** (*str*) – the data source to retrieve. Must be one of: ‘eia860’, ‘eia923’, ‘ferc1’, or ‘epacems’.
- **year** (*int or None*) – the year of data that the returned path should pertain to. Must be within the range of valid data years, which is specified for each data source in pudl.constants.data_years. Note that for data (like EPA CEMS) that have multiple datasets per year, this function will download all the files for the specified year. Use None for data sources that do not have multiple years.
- **states** (*iterable*) – List of two letter US state abbreviations indicating which states data should be downloaded for.
- **data_dir** (*path-like*) – Path to the top level datastore directory.
- **clobber** (*bool*) – If True, clobber the existing file and note that the file will need to be replaced with an updated file.

Returns Whether an update is needed (True) or not (False)

Return type `bool`

`pudl.workspace.datastore.download(source, year, states, data_dir)`

Download the original data for the specified data source and year.

Given a data source and the desired year of data, download the original data files from the appropriate federal website, and place them in a temporary directory within the data store. This function does not do any checking to see whether the file already exists, or needs to be updated, and does not do any of the organization of the datastore after download, it simply gets the requested file.

Parameters

- **source** (*str*) – the data source to retrieve. Must be one of: ‘eia860’, ‘eia923’, ‘ferc1’, or ‘epacems’.
- **year** (*int or None*) – the year of data that the returned path should pertain to. Must be within the range of valid data years, which is specified for each data source in `pudl.constants.data_years`. Note that for data (like EPA CEMS) that have multiple datasets per year, this function will download all the files for the specified year. Use `None` for data sources that do not have multiple years.
- **states** (*iterable*) – List of two letter US state abbreviations indicating which states data should be downloaded for.
- **data_dir** (*path-like*) – Path to the top level datastore directory.

Returns The path to the local downloaded file.

Return type path-like

Todo: Return to

`pudl.workspace.datastore.organize(source, year, states, data_dir, unzip=True, dl=True)`
Put downloaded original data file where it belongs in the datastore.

Once we’ve downloaded an original file from the public website it lives on we need to put it where it belongs in the datastore. Optionally, we also unzip it and clean up the directory hierarchy that results from unzipping.

Parameters

- **source** (*str*) – the data source to retrieve. Must be one of: ‘eia860’, ‘eia923’, ‘ferc1’, or ‘epacems’.
- **year** (*int or None*) – the year of data that the returned path should pertain to. Must be within the range of valid data years, which is specified for each data source in `pudl.constants.data_years`. Use `None` for data sources that do not have multiple years.
- **data_dir** (*path-like*) – Path to the top level datastore directory.
- **unzip** (*bool*) – If `True`, unzip the file once downloaded, and place the resulting data files where they ought to be in the datastore.
- **dl** (*bool*) – If `False`, the files were not downloaded in this run.

Returns `None`

Todo: Replace 4 assert statements

`pudl.workspace.datastore.parallel_update(sources, years_by_source, states, data_dir, clobber=False, unzip=True, dl=True)`
Download many original source data files in parallel using threads.

`pudl.workspace.datastore.path(source, data_dir, year=None, month=None, state=None, file=True)`
Construct a variety of local datastore paths for a given data source.

PUDL expects the original data it ingests to be organized in a particular way. This function allows you to easily construct useful paths that refer to various parts of the data store, by specifying the data source you are interested in, and optionally the year of data you're seeking, as well as whether you want the originally downloaded files for that year, or the directory in which a given year's worth of data for a particular data source can be found.

Note: if you change the default arguments here, you should also change them for `paths_for_year()`

Parameters

- **source** (*str*) – A string indicating which data source we are going to be downloading. Currently it must be one of the following: - 'ferc1' - 'eia923' - 'eia860' - 'epacems'
- **data_dir** (*path-like*) – Path to the top level datastore directory.
- **year** (*int or None*) – the year of data that the returned path should pertain to. Must be within the range of valid data years, which is specified for each data source in `pudl.constants.data_years`, unless year is set to zero, in which case only the top level directory for the data source specified in source is returned. If None, no subdirectory is used for the data source.
- **month** (*int*) – Month of year (1-12). Only applies to epacems.
- **state** (*str*) – Two letter US state abbreviation. Only applies to epacems.
- **file** (*bool*) – If True, return the full path to the originally downloaded file specified by the data source and year. If file is true, year must not be set to zero, as a year is required to specify a particular downloaded file.

Returns the path to requested resource within the local PUDL datastore.

Return type *str*

```
pudl.workspace.datastore.paths_for_year(source, data_dir, year=None, states=None,
                                         file=True)
```

Derive all paths for a given source and year. See `path()` for details.

Parameters

- **source** (*str*) – A string indicating which data source we are going to be downloading. Currently it must be one of the following: - 'ferc1' - 'eia923' - 'eia860' - 'epacems'
- **data_dir** (*path-like*) – Path to the top level datastore directory.
- **year** (*int or None*) – the year of data that the returned path should pertain to. Must be within the range of valid data years, which is specified for each data source in `pudl.constants.data_years`, unless year is set to zero, in which case only the top level directory for the data source specified in source is returned. If None, no subdirectory is used for the data source.
- **month** (*int*) – Month of year (1-12). Only applies to epacems.
- **state** (*str*) – Two letter US state abbreviation. Only applies to epacems.
- **file** (*bool*) – If True, return the full path to the originally downloaded file specified by the data source and year. If file is true, year must not be set to zero, as a year is required to specify a particular downloaded file.

Returns the path to requested resource within the local PUDL datastore.

Return type *str*

```
pudl.workspace.datastore.source_url(source, year, month=None, state=None, table=None)
```

Construct a download URL for the specified federal data source and year.

Parameters

- **source** (*str*) – A string indicating which data source we are going to be downloading. Currently it must be one of the following: - ‘eia860’ - ‘eia861’ - ‘eia923’ - ‘ferc1’ - ‘epacems’
- **year** (*int or None*) – the year for which data should be downloaded. Must be within the range of valid data years, which is specified for each data source in the pudl.constants module. Use None for data sources that do not have multiple years.
- **month** (*int*) – the month for which data should be downloaded. Only used for EPA CEMS.
- **state** (*str*) – the state for which data should be downloaded. Only used for EPA CEMS.
- **table** (*str*) – the table for which data should be downloaded. Only used for EPA IPM.

Returns a full URL from which the requested data may be obtained

Return type download_url (*str*)

```
pudl.workspace.datastore.update(source, year, states, data_dir, clobber=False, unzip=True,
                                dl=True)
```

Update the local datastore for the given source and year.

If necessary, pull down a new copy of the data for the specified data source and year. If we already have the requested data, do nothing, unless clobber is True – in which case remove the existing data and replace it with a freshly downloaded copy.

Note that update_datastore.py runs this function in parallel, so files multiple sources and years may be in progress simultaneously.

Parameters

- **source** (*str*) – the data source to retrieve. Must be one of: ‘eia860’, ‘eia923’, ‘ferc1’, or ‘epacems’.
- **year** (*int*) – the year of data that the returned path should pertain to. Must be within the range of valid data years, which is specified for each data source in pudl.constants.data_years.
- **states** (*iterable*) – List of two letter US state abbreviations indicating which states data should be downloaded for. Currently only affects the epacems dataset.
- **clobber** (*bool*) – If true, replace existing copy of the requested data if we have it, with freshly downloaded data.
- **unzip** (*bool*) – If true, unzip the file once downloaded, and place the resulting data files where they ought to be in the datastore. EPA CEMS files will never be unzipped.
- **data_dir** (*str*) – The data directory which holds the PUDL datastore.
- **dl** (*bool*) – If False, don’t download the files, only unzip ones that are already present. If True, do download the files. Either way, still obey the unzip and clobber settings. (unzip=False and dl=False will do nothing.)

Returns None

pudl.workspace.datastore_cli module

A CLI for fetching public utility data from reporting agency servers.

This script will generate a datastore on a datastore directory. By default, the directory will end up in wherever you have designated “PUDL_IN” in the settings file \$HOME/.pudl.yml. You can use this script to specific only specific datasets to download, only specific years or states but by default, it will grab everything. A populated datastore is required to use other PUDL tools, like the ETL script (*pudl_etl*) and all of the post-ETL processes.

`pudl.workspace.datastore_cli.main()`

Manage and update the PUDL datastore.

`pudl.workspace.datastore_cli.parse_command_line(argv)`

Parse command line arguments. See the `-h` option for more details.

Parameters `argv` (*str*) – Command line arguments, which must include caller filename.

Returns Dictionary of command line arguments and their parsed values.

Return type `dict`

pudl.workspace.setup module

Tools for setting up and managing PUDL workspaces.

`pudl.workspace.setup.deploy(pkg_path, deploy_dir, ignore_files, clobber=False)`

Deploy all files from a `package_data` directory into a workspace.

Parameters

- **pkg_path** (*str*) – Dotted module path to the subpackage inside of `package_data` containing the resources to be deployed.
- **deploy_dir** (*os.PathLike*) – Directory on the filesystem to which the files within `pkg_path` should be deployed.
- **ignore_files** (*iterable*) – List of filenames (strings) that may be present in the `pkg_path` subpackage, but that should be ignored.
- **clobber** (*bool*) – if True, replace existing copies of the files that are being deployed from `pkg_path` to `deploy_dir`. If False, do not replace existing files.

Returns None

`pudl.workspace.setup.derive_paths(pudl_in, pudl_out)`

Derive PUDL paths based on given input and output paths.

If no configuration file path is provided, attempt to read in the user configuration from a file called `.pudl.yml` in the user's HOME directory. Presently the only values we expect are `pudl_in` and `pudl_out`, directories that store files that PUDL either depends on that rely on PUDL.

Parameters

- **pudl_in** (*os.PathLike*) – Path to the directory containing the PUDL input files, most notably the data directory which houses the raw data downloaded from public agencies by the `pudl.workspace.datastore` tools. `pudl_in` may be the same directory as `pudl_out`.
- **pudl_out** (*os.PathLike*) – Path to the directory where PUDL should write the outputs it generates. These will be organized into directories according to the output format (sqlite, datapackage, etc.).

Returns

A dictionary containing common PUDL settings, derived from those read out of the YAML file. Mostly paths for inputs & outputs.

Return type `dict`

`pudl.workspace.setup.get_defaults()`

Read paths to default PUDL input/output dirs from user's `$HOME/.pudl.yml`.

Parameters None –

Returns The contents of the user’s PUDL settings file, with keys `pudl_in` and `pudl_out` defining their default PUDL workspace. If the `$HOME/.pudl.yml` file does not exist, set these paths to None.

Return type `dict`

`pudl.workspace.setup.init(pudl_in, pudl_out, clobber=False)`
Set up a new PUDL working environment based on the user settings.

Parameters

- **pudl_in** (`os.PathLike`) – Path to the directory containing the PUDL input files, most notably the data directory which houses the raw data downloaded from public agencies by the `pudl.workspace.datastore` tools. `pudl_in` may be the same directory as `pudl_out`.
- **pudl_out** (`os.PathLike`) – Path to the directory where PUDL should write the outputs it generates. These will be organized into directories according to the output format (sqlite, datapackage, etc.).
- **clobber** (`bool`) – if True, replace existing files. If False (the default) do not replace existing files.

Returns None

`pudl.workspace.setup.set_defaults(pudl_in, pudl_out, clobber=False)`
Set default user input and output locations in `$HOME/.pudl.yml`.

Create a user settings file for future reference, that defines the default PUDL input and output directories. If this file already exists, behavior depends on the `clobber` parameter, which is False by default. If it’s True, the existing file is replaced. If False, the existing file is not changed.

Parameters

- **pudl_in** (`os.PathLike`) – Path to be used as the default input directory for PUDL – this is where `pudl.workspace.datastore` will look to find the data directory, full of data from public agencies.
- **pudl_out** (`os.PathLike`) – Path to the default output directory for PUDL, where results of data processing will be organized.
- **clobber** (`bool`) – If True and a user settings file exists, overwrite it. If False, do not alter the existing file. Defaults to False.

Returns None

pudl.workspace.setup_cli module

Set up a well-organized PUDL workspace.

This script creates a well-defined directory structure for use by the PUDL package, and copies several example settings files and Jupyter notebooks into it to get you started. If the command is run without any arguments, it will create this workspace in your current directory.

The script will also create a file named `.pudl.yml`, describing the location of your PUDL workspace. The PUDL package will refer to this location in the future to know where it should look for raw data, where to put its outputs, etc. This file can be edited to change the default input and output directories if you wish. However, make sure those workspaces are set up using this script.

It's also possible to specify different input and output directories, which is useful if you want to use a single PUDL data store (which may contain many GB of data) to support several different workspaces. See the `--pudl_in` and `--pudl_out` options.

By default the script will not overwrite existing files. If you want it to replace existing files (including your `.pudl.yml` file which defines your default PUDL workspace) use the `--lobber` option.

The directory structure set up for PUDL looks like this:

PUDL_IN

└─ **data** ── eia ── form860 ── form923 ── epa ── cems ── ipm ── ferc ── form1 ──
 tmp

PUDL_OUT ── datapackage ── environment.yml ── notebook ── parquet ── settings ── sqlite

Initially, the directories in the data store will be empty. The `pudl_data` or `pudl_etl` commands will download data from public sources and organize it for you there by source. The PUDL_OUT directories are organized by the type of file they contain.

```
pudl.workspace.setup_cli.initialize_parser()  
    Parse command line arguments for the pudl_setup script.
```

```
pudl.workspace.setup_cli.main()  
    Set up a new default PUDL workspace.
```

Module contents

Tools for acquiring PUDL's original input data and organizing it locally.

The datastore subpackage takes care of downloading original data from various public sources, organizing it locally, and providing a programmatic interface to that collection of raw inputs, which we refer to as the PUDL datastore.

These tools are available both as a library module, and via a command line interface installed as an entrypoint script called `pudl_data`. For full reproducibility of PUDL's ETL pipeline outputs, the datastore should be archived alongside the PUDL release which was used and the resulting datapackage outputs.

Submodules

pudl.cli module

A command line interface (CLI) to the main PUDL ETL functionality.

This script generates datapackages based on the datapackage settings enumerated in the `settings_file` which is given as an argument to this script. If the settings has empty datapackage parameters (meaning there are no years or tables included), no datapackages will be generated. If the settings include a datapackage that has empty parameters, the other valid datapackages will be generated, but not the empty one. If there are invalid parameters (meaning a year that is not included in the `pudl.constant.working_years`), the build will fail early on in the process.

The datapackages will be stored in "PUDL_OUT" in the "datapackage" subdirectory. Currently, this function only uses default directories for "PUDL_IN" and "PUDL_OUT" (meaning those stored in `$HOME/.pudl.yml`). To setup your default pudl directories see the `pudl_setup` script (`pudl_setup --help` for more details).

```
pudl.cli.main()  
    Parse command line and initialize PUDL DB.  
  
pudl.cli.parse_command_line(argv)  
    Parse script command line arguments. See the -h option.
```

Parameters `argv` (*list*) – command line arguments including caller file name.

Returns A dictionary mapping command line arguments to their values.

Return type `dict`

pudl.constants module

A warehouse for constant values required to initialize the PUDL Database.

This constants module stores and organizes a bunch of constant values which are used throughout PUDL to populate static lists within the data packages or for data cleaning purposes.

`pudl.constants.aer_coal_strings = ['col', 'woc', 'pc']`

A list of EIA 923 AER fuel type strings associated with coal.

Type `list`

`pudl.constants.aer_fuel_type_strings = {'coal': ['col', 'woc', 'pc'], 'gas': ['mlg', 'ng']}`

A dictionary mapping EIA 923 AER fuel types (keys) to lists of strings associated with that fuel type (values).

Type `dict`

`pudl.constants.aer_gas_strings = ['mlg', 'ng', 'oog']`

A list of EIA 923 AER fuel type strings associated with gas.

Type `list`

`pudl.constants.aer_hydro_strings = ['hps', 'hyc']`

A list of EIA 923 AER fuel type strings associated with hydro power.

Type `list`

`pudl.constants.aer_nuclear_strings = ['nuc']`

A list of EIA 923 AER fuel type strings associated with nuclear power.

Type `list`

`pudl.constants.aer_oil_strings = ['dfo', 'rfo', 'woo']`

A list of EIA 923 AER fuel type strings associated with oil.

Type `list`

`pudl.constants.aer_other_strings = ['geo', 'orw', 'oth']`

A list of EIA 923 AER fuel type strings associated with other fuel.

Type `list`

`pudl.constants.aer_solar_strings = ['sun']`

A list of EIA 923 AER fuel type strings associated with solar power.

Type `list`

`pudl.constants.aer_waste_strings = ['www']`

A list of EIA 923 AER fuel type strings associated with waste.

Type `list`

`pudl.constants.aer_wind_strings = ['wnd']`

A list of EIA 923 AER fuel type strings associated with wind power.

Type `list`

`pudl.constants.base_data_urls = {'eia860': 'https://www.eia.gov/electricity/data/eia860'}`

A dictionary containing data sources (keys) and their base data URLs (values).

Type dict

`pudl.constants.boiler_fuel_map_eia923 = plant_id_eia ... report_year year_index ... 2009 p`
A DataFrame of metadata from EIA 923 Boiler Fuel.

Type pandas.DataFrame

`pudl.constants.boiler_generator_assn_map_eia860 = utility_id_eia plant_id_eia boiler_id ge`
A DataFrame of metadata from EIA 860 Boiler Generator Association.

Type pandas.DataFrame

`pudl.constants.canada_prov_terr = {'AB': 'Alberta', 'BC': 'British Columbia', 'CN': 'Canada`
A dictionary containing Canadian provinces' and territories' abbreviations (keys) and names (values)

Type dict

`pudl.constants.cems_states = {'AL': 'Alabama', 'AR': 'Arkansas', 'AZ': 'Arizona', 'CA': 'Ca`
A dictionary containing US state abbreviations (keys) and names (values) that are present in the CEMS dataset

Type dict

`pudl.constants.census_region = {'ENC': 'East North Central', 'ESC': 'East South Central',`
A dictionary mapping Census Region abbreviations (keys) to Census Region names (values).

Type dict

`pudl.constants.coalmine_country_eia923 = {'AU': 'AUS', 'CL': 'COL', 'CN': 'CAN', 'IM': 'unl`
A dictionary mapping coal mine country codes (keys) to ISO-3166-1 three letter country codes (values).

Type dict

`pudl.constants.coalmine_type_eia923 = {'P': 'Preparation Plant', 'S': 'Surface', 'SU': 'Bot`
A dictionary mapping EIA 923 coal mine type codes (keys) to descriptions (values).

Type dict

`pudl.constants.contract_type_eia923 = {'C': 'Contract - Fuel received under a purchase orde`
A dictionary mapping EIA 923 contract codes (keys) to contract descriptions (values) for each month in the Fuel Receipts and Costs table.

Type dict

`pudl.constants.contributors = {'alana-wilson': {'email': 'alana.wilson@catalyst.coop', 'c`
A dictionary of dictionaries containing organization names (keys) and their attributes (values).

Type dict

`pudl.constants.contributors_by_source = {'eia860': ['catalyst-cooperative', 'zane-selvans`
A dictionary of data sources (keys) and lists of contributors (values).

Type dict

`pudl.constants.cpi_diesel_strings = ['DIESEL', 'Diesel Engine', 'Diesel Turbine']`
A list of strings for fuel type diesel compiled by Climate Policy Initiative.

Type list

`pudl.constants.cpi_geothermal_strings = ['Steam - Geothermal']`
A list of strings for fuel type geothermal compiled by Climate Policy Initiative.

Type list

`pudl.constants.cpi_natural_gas_strings = ['Combined Cycle', 'Combustion Turbine', 'GT', 'G`
A list of strings for fuel type gas compiled by Climate Policy Initiative.

Type list


```
pudl.constants.cpi_nuclear_strings = ['Nuclear', 'Nuclear (3)']
```

A list of strings for fuel type nuclear compiled by Climate Policy Initiative.

Type `list`

```
pudl.constants.cpi_other_strings = ['IC', 'Internal Combustion', 'Int Combust - Note 1', 'I
```

A list of strings for fuel type other compiled by Climate Policy Initiative.

Type `list`

```
pudl.constants.cpi_plant_kind_strings = {'diesel': ['DIESEL', 'Diesel Engine', 'Diesel Tur
```

A dictionary linking fuel types (keys) to lists of strings associated by Climate Policy Institute with those fuel types (values).

Type `dict`

```
pudl.constants.cpi_solar_strings = ['Solar Photovoltaic', 'Solar Thermal', 'SOLAR PROJECT']
```

A list of strings for fuel type photovoltaic compiled by Climate Policy Initiative.

Type `list`

```
pudl.constants.cpi_steam_strings = ['Steam', 'Steam Units 1, 2, 3', 'Resp Share St Note 3']
```

A list of strings for fuel type steam compiled by Climate Policy Initiative.

Type `list`

```
pudl.constants.cpi_wind_strings = ['Wind', 'Wind Turbine', 'Wind - Turbine', 'Wind Energy']
```

A list of strings for fuel type wind compiled by Climate Policy Initiative.

Type `list`

```
pudl.constants.data_source_info = {'eia860': {'path': 'https://www.eia.gov/electricity/d
```

A dictionary of dictionaries containing datasources (keys) and associated attributes (values)

Type `dict`

```
pudl.constants.data_sources = ('eia860', 'eia923', 'epacems', 'ferc1', 'epaipm')
```

A tuple containing the data sources we are able to pull into PUDL.

Type `tuple`

```
pudl.constants.data_years = {'eia860': (2001, 2002, 2003, 2004, 2005, 2006, 2007, 2008, 20
```

A dictionary of data sources (keys) and tuples containing the years that we expect to be able to download for each data source (values).

Type `dict`

```
pudl.constants.dbf_typemap = {'+': 'XXX', '0': <class 'sqlalchemy.sql.sqltypes.Integer'>
```

A dictionary mapping field types in the DBF objects (keys) to the corresponding generic SQLAlchemy Column types.

Type `dict`

```
pudl.constants.eia860_pudl_tables = ('boiler_generator_assn_eia860', 'utilities_eia860', 'p
```

A tuple containing the list of EIA 860 tables that can be successfully pulled into PUDL.

Type `tuple`

```
pudl.constants.eia923_pudl_tables = ('generation_fuel_eia923', 'boiler_fuel_eia923', 'gene
```

A tuple containing the EIA923 tables that can be successfully integrated into PUDL.

Type `tuple`

```
pudl.constants.energy_source_eia923 = {'ANT': 'Anthracite Coal', 'BFG': 'Blast Furnace Gas
```

A dictionary mapping fuel codes (keys) to fuel descriptions (values) for each fuel receipt from the EIA 923 Fuel Receipts and Costs table.

Type dict

`pudl.constants.energy_source_eia_simple_map` = {'coal': ['ANT', 'BIT', 'LIG', 'PC', 'SUB'],
A dictionary mapping EIA fuel types (keys) to fuel codes (values).

Type dict

`pudl.constants.entities` = {'boilers': [['plant_id_eia', 'boiler_id'], ['prime_mover_code'],
A dictionary containing table name strings (keys) and lists of columns to keep for those tables (values).

Type dict

`pudl.constants.entity_tables` = ['utilities_entity_eia', 'plants_entity_eia', 'generators_eia',
A list of PUDL entity tables.

Type list

`pudl.constants.epacems_additional_plant_info_file` = <_io.TextIOWrapper name='/home/docs/ch...
typing.TextIO:

Todo: Return to

`pudl.constants.epacems_columns_fill_na_dict` = {'gross_load_mw': 0.0, 'heat_content_mmbtu':
the set of EPA CEMS columns to

Todo: Return to

Type set

`pudl.constants.epacems_columns_to_ignore` = {'CO2_RATE', 'CO2_RATE (tons/mmBtu)', 'CO2_RATE',
The set of EPA CEMS columns to ignore when reading data.

Type set

`pudl.constants.epacems_csv_dtypes` = {'CO2_MASS': <class 'float'>, 'CO2_MASS (tons)': <class
A dictionary containing column names (keys) and data types (values) for EPA CEMS.

Type dict

`pudl.constants.epacems_rename_dict` = {'CO2_MASS': 'co2_mass_tons', 'CO2_MASS (tons)': 'co2
A dictionary containing EPA CEMS column names (keys) and replacement names to use when reading those
columns into PUDL (values).

Type dict

`pudl.constants.epacems_tables` = 'hourly_emissions_epacems'
A tuple containing tables of EPA CEMS data to pull into PUDL.

Type tuple

`pudl.constants.epaipm_pudl_tables` = ('transmission_single_epaipm', 'transmission_joint_epa
A tuple containing the EPA IPM tables that can be successfully integrated into PUDL.

Type tuple

`pudl.constants.epaipm_region_aggregations` = {'ISONE': ['NENG_CT', 'NENGREST', 'NENG_ME'],
A dictionary containing EPA IPM regions (keys) and lists of their associated abbreviations (values).

Type dict

`pudl.constants.epaipm_region_names = ['ERC_PHDL', 'ERC_REST', 'ERC_FRNT', 'ERC_GWAY', 'ERC_`
 A list of EPA IPM region names.

Type `list`

`pudl.constants.epaipm_url_ext = {'load_curves_epaipm': 'table_2-2_load_duration_curves_us`
 A dictionary of EPA IPM tables and associated URLs extensions for downloading that table's data.

Type `dict`

`pudl.constants.ferc1_1kgal_strings = ['oil(1000 gal)', 'oil(1000)', 'oil (1000)', 'oil(1000`
 A list of fuel unit strings for thousand gallons.

Type `list`

`pudl.constants.ferc1_bbl_strings = ['barrel', 'bbls', 'bbl', 'barrels', 'bbrl', 'bbl.', 'bl`
 A list of fuel unit strings for barrels.

Type `list`

`pudl.constants.ferc1_coal_strings = ['coal', 'coal-subbit', 'lignite', 'coal(sb)', 'coal (s`
 A list of strings which are used to represent coal fuel in FERC Form 1 reporting.

Type `list`

`pudl.constants.ferc1_const_type_conventional = ['conventional', 'conventional', 'convention`
 A list of strings from FERC Form 1 associated with the conventional construction type.

Type `list`

`pudl.constants.ferc1_const_type_outdoor = ['outdoor', 'outdoor boiler', 'full outdoor', 'ov`
 A list of strings from FERC Form 1 associated with the outdoor construction type.

Type `list`

`pudl.constants.ferc1_const_type_strings = {'conventional': ['conventional', 'conventional`
 A dictionary of construction types (keys) and lists of construction type strings associated with each type (values) from FERC Form 1.

Type `dict`

`pudl.constants.ferc1_data_tables = ('f1_acb_epda', 'f1_accumdepr_prvsn', 'f1_accumdfrrdtax`
 A tuple containing the FERC Form 1 tables that have the same composite primary keys: [respondent_id, report_year, report_prd, row_number, spplmnt_num].

Type `tuple`

`pudl.constants.ferc1_dbf2tbl = {'F1_1': 'f1_respondent_id', 'F1_10': 'f1_allowances', 'F`
 A dictionary mapping FERC Form 1 DBF files (w/o .DBF file extension) (keys) to database table names (values).

Type `dict`

`pudl.constants.ferc1_default_tables = ('f1_respondent_id', 'f1_fuel', 'f1_steam', 'f1_gnrt`
 A tuple containing the FERC Form 1 columns PUDL is initially focused on.

Type `tuple`

`pudl.constants.ferc1_fuel_strings = {'coal': ['coal', 'coal-subbit', 'lignite', 'coal(sb)`
 A dictionary linking fuel types (keys) to lists of various strings representing that fuel (values)

Type `dict`

`pudl.constants.ferc1_fuel_unit_strings = {'1kgal': ['oil(1000 gal)', 'oil(1000)', 'oil (1`
 A dictionary linking fuel units (keys) to lists of various strings representing those fuel units (values)

Type `dict`

`pudl.constants.ferc1_gal_strings = ['gallons', 'gal.', 'gals', 'gals.', 'gallon', 'gal']`
 A list of fuel unit strings for gallons.

Type `list`

`pudl.constants.ferc1_gas_strings = ['gas', 'gass', 'methane', 'natural gas', 'blast gas',`
 A list of strings which are used to represent gas fuel in FERC Form 1 reporting.

Type `list`

`pudl.constants.ferc1_gramsU_strings = ['gram', 'grams', 'gm u', 'grams u235', 'grams u-235']`
 A list of fuel unit strings for grams.

Type `list`

`pudl.constants.ferc1_huge_tables = {'f1_footnote_data', 'f1_footnote_tbl', 'f1_note_fin_str'`
 A set containing large FERC Form 1 tables.

Type `set`

`pudl.constants.ferc1_kgU_strings = ['kg of uranium', 'kg uranium', 'kilg. u-235', 'kg u-235']`
 A list of fuel unit strings for thousand grams.

Type `list`

`pudl.constants.ferc1_mcf_strings = ['mcf', 'mcf's', 'mcfs', 'mcf.', 'gas mcf', '"gas" mcf']`
 A list of fuel unit strings for thousand cubic feet.

Type `list`

`pudl.constants.ferc1_mmbtu_strings = ['mmbtu', 'mmbtus', 'mmbtu's', 'nuclear-mmbtu', 'nucle'`
 A list of fuel unit strings for million British Thermal Units.

Type `list`

`pudl.constants.ferc1_mwdth_strings = ['mwd therman', 'mw days-therm', 'mwd thrml', 'mwd the'`
 A list of fuel unit strings for megawatt days thermal.

Type `list`

`pudl.constants.ferc1_mwhth_strings = ['mwh them', 'mwh threm', 'nwh therm', 'mwhth', 'mwh t'`
 A list of fuel unit strings for megawatt hours thermal.

Type `list`

`pudl.constants.ferc1_nuke_strings = ['nuclear', 'grams of uran', 'grams of', 'grams of ura'`
 A list of strings which are used to represent nuclear fuel in FERC Form 1 reporting.

Type `list`

`pudl.constants.ferc1_oil_strings = ['oil', '#6 oil', '#2 oil', 'fuel oil', 'jet', 'no. 2 o'`
 A list of strings which are used to represent oil fuel in FERC Form 1 reporting.

Type `list`

`pudl.constants.ferc1_other_strings = ['steam', 'purch steam', 'purch. steam', 'other', 'con'`
 A list of strings which are used to represent other fuels in FERC Form 1 reporting.

Type `list`

`pudl.constants.ferc1_plant_kind_combined_cycle = ['Combined cycle', 'combined cycle', 'com'`
 A list of strings from FERC Form 1 for the combined cycle plant kind.

Type `list`

`pudl.constants.ferc1_plant_kind_combustion_turbine = ['combustion turbine', 'gt', 'gas tur'`
 A list of strings from FERC Form 1 for the combustion turbine plant kind.

Type `list`

`pudl.constants.ferc1_plant_kind_geothermal = ['steam - geothermal', 'steam_geothermal']`
 A list of strings from FERC Form 1 for the geothermal plant kind.

Type `list`

`pudl.constants.ferc1_plant_kind_nuke = ['nuclear', 'nuclear (3)']`
 A list of strings from FERC Form 1 for the nuclear plant kind.

Type `list`

`pudl.constants.ferc1_plant_kind_photovoltaic = ['solar photovoltaic', 'photovoltaic']`
 A list of strings from FERC Form 1 for the photovoltaic plant kind.

Type `list`

`pudl.constants.ferc1_plant_kind_solar_thermal = ['solar thermal']`
 A list of strings from FERC Form 1 for the solar thermal plant kind.

Type `list`

`pudl.constants.ferc1_plant_kind_steam_turbine = ['coal', 'steam', 'steam units 1 2 3', 'steam units 4 5 6']`
 A list of strings from FERC Form 1 for the steam turbine plant kind.

Type `list`

`pudl.constants.ferc1_plant_kind_strings = {'combined_cycle': ['Combined cycle', 'combined cycle']}`
 A dictionary of plant kinds (keys) and associated lists of plant_fuel strings (values).

Type `dict`

`pudl.constants.ferc1_plant_kind_wind = ['wind', 'wind energy', 'wind turbine', 'wind - turbine']`
 A list of strings from FERC Form 1 for the wind plant kind.

Type `list`

`pudl.constants.ferc1_power_purchase_type = {'AD': 'adjustment', 'EX': 'electricity exchange'}`
 A dictionary of abbreviations (keys) and types (values) for power purchase agreements from FERC Form 1.

Type `dict`

`pudl.constants.ferc1_pudl_tables = ('fuel_ferc1', 'plants_steam_ferc1', 'plants_small_ferc1')`
 A tuple containing the FERC Form 1 tables that can be successfully integrated into PUDL.

Type `tuple`

`pudl.constants.ferc1_tbl2dbf = {'f1_106_2009': 'F1_106_2009', 'f1_106a_2009': 'F1_106A_2009'}`
 A dictionary mapping database table names (keys) to FERC Form 1 DBF files (w/o .DBF file extension) (values).

Type `dict`

`pudl.constants.ferc1_ton_strings = ['toms', 'taons', 'tones', 'col-tons', 'toncoaleq', 'coal-tons']`
 A list of fuel unit strings for tons.

Type `list`

`pudl.constants.ferc1_waste_strings = ['tires', 'tire', 'refuse', 'switchgrass', 'wood waste']`
 A list of strings which are used to represent waste fuel in FERC Form 1 reporting.

Type `list`

`pudl.constants.ferc_1_plant_kind_internal_combustion = ['ic', 'internal combustion', 'diesel']`
 A list of strings from FERC Form 1 for the internal combustion plant kind.

Type `list`

`pudl.constants.ferc_accumulated_depreciation = row_number ... ferc_account_description 0 1`
 A list of tuples containing row numbers, FERC account IDs, and FERC account descriptions from FERC Form 1 page 219, Accumulated Provision for Depreciation of electric utility plant (Account 108).

Type `list`

`pudl.constants.ferc_electric_plant_accounts = row_number ... ferc_account_description 0 2.`
 A list of tuples containing row numbers, FERC account IDs, and FERC account descriptions from FERC Form 1 pages 204-207, Electric Plant in Service.

Type `list`

`pudl.constants.file_pages_eia860 = {'enviro_assn': ['boiler_generator_assn'], 'generators`
 A dictionary containing file names (keys) and lists of tab names to read (values) for EIA 860.

Type `dict`

`pudl.constants.files_dict_eia860 = {'enviro_assn': '*EnviroAssoc*', 'envrio_equipment':`
 A dictionary containing file names (keys) and file name patterns to glob (values) for EIA 860.

Type `dict`

`pudl.constants.files_dict_epaipm = {'load_curves_epaipm': '*table_2-2_*', 'plant_region_ma`
 A dictionary of EPA IPM tables and strings that files of those tables contain.

Type `dict`

`pudl.constants.files_eia860 = ('enviro_assn', 'utilities', 'plants', 'generators', 'ownersl`
 A tuple containing EIA 860 file names.

Type `tuple`

`pudl.constants.fuel_group_eia923 = ('coal', 'natural_gas', 'petroleum', 'petroleum_coke',`
 A tuple containing EIA 923 fuel groups.

Type `tuple`

`pudl.constants.fuel_group_eia923_simple_map = {'coal': ['coal', 'petroleum coke'], 'gas':`
 A dictionary mapping EIA 923 simple fuel types (“oil”, “coal”, “gas”) (keys) to fuel types (values).

Type `dict`

`pudl.constants.fuel_receipts_costs_map_eia923 = report_year report_month ... moisture_cont`
 A DataFrame of metadata from EIA 923 Fuel Receipts and Costs.

Type `pandas.DataFrame`

`pudl.constants.fuel_type_aer_eia923 = {'COL': 'Coal', 'DFO': 'Distillate Petroleum', 'GEO'`
 A dictionary mapping EIA 923 AER fuel types (keys) to lists of strings associated with that fuel type (values).

Type `dict`

`pudl.constants.fuel_type_eia860_coal_strings = ['ant', 'bit', 'cbl', 'lig', 'pc', 'rc', 's`
 A list of strings from EIA 860 associated with fuel type coal.

Type `list`

`pudl.constants.fuel_type_eia860_gas_strings = ['bfg', 'lfg', 'mlg', 'ng', 'obg', 'og', 'pg`
 A list of strings from EIA 860 associated with fuel type gas.

Type `list`

`pudl.constants.fuel_type_eia860_hydro_strings = ['wat', 'hyc', 'hps', 'hydro']`
 A list of strings from EIA 860 associated with hydro power.

Type `list`

A list of strings from EIA 860 associated with nuclear power.

A list of strings from EIA 860 associated with fuel type oil.

A list of strings from EIA 860 associated with fuel type other.

A dictionary mapping EIA 860 fuel types (keys) to lists of strings associated with that fuel type (values).

A list of strings from EIA 860 associated with solar power.

A list of strings from EIA 860 associated with fuel type waste.

• •

A list of strings from EIA 860 associated with wind power.

A dictionary mapping EIA 923 fuel type codes (keys) and fuel type names/descriptions (values).

— **11** —

11

A list of strings from EIA 923 Boiler Fuel associated with fuel type coal.

• • •

A list of strings from EIA 923 Boiler Fuel associated with fuel type gas.

A list of strings from EIA 923 Boiler Fuel associated with fuel type oil.

—

A list of strings from EIA 923 Boiler Fuel associated with fuel type other.

62

A dictionary mapping EIA 923 Boiler Fuel fuel types (keys) to lists of strings associated with that fuel type

y	π	φ	$y_1 = (\cdot \cdot y)$	φ	y_1
(values)					

()

62

`pudl.constants.fuel_type_eia923_boiler_fuel_waste_strings = ['ab', 'blq', 'msb', 'msn', 'ol']`
 A list of strings from EIA 923 Boiler Fuel associated with fuel type waste.

Type `list`

`pudl.constants.fuel_type_eia923_gen_fuel_coal_strings = ['ant', 'bit', 'cbl', 'lig', 'pc', 'pco']`
 The list of EIA 923 Generation Fuel strings associated with coal fuel.

Type `list`

`pudl.constants.fuel_type_eia923_gen_fuel_gas_strings = ['bfg', 'lfg', 'ng', 'og', 'obg', 'pbg']`
 The list of EIA 923 Generation Fuel strings associated with gas fuel.

Type `list`

`pudl.constants.fuel_type_eia923_gen_fuel_hydro_strings = ['wat']`
 The list of EIA 923 Generation Fuel strings associated with hydro power.

Type `list`

`pudl.constants.fuel_type_eia923_gen_fuel_nuclear_strings = ['nuc']`
 The list of EIA 923 Generation Fuel strings associated with nuclear power.

Type `list`

`pudl.constants.fuel_type_eia923_gen_fuel_oil_strings = ['dfo', 'rfo', 'wo', 'jfb', 'ker']`
 The list of EIA 923 Generation Fuel strings associated with oil fuel.

Type `list`

`pudl.constants.fuel_type_eia923_gen_fuel_other_strings = ['geo', 'mwh', 'oth', 'pur', 'wh']`
 The list of EIA 923 Generation Fuel strings associated with geothermal power.

Type `list`

`pudl.constants.fuel_type_eia923_gen_fuel_simple_map = {'coal': ['ant', 'bit', 'cbl', 'lig', 'pco'], 'gas': ['bfg', 'lfg', 'ng', 'og', 'obg', 'pbg'], 'hydro': ['wat'], 'nuclear': ['nuc'], 'oil': ['dfo', 'rfo', 'wo', 'jfb', 'ker'], 'other': ['geo', 'mwh', 'oth', 'pur', 'wh']}`
 A dictionary mapping EIA 923 Generation Fuel fuel types (keys) to lists of strings associated with that fuel type (values).

Type `dict`

`pudl.constants.fuel_type_eia923_gen_fuel_solar_strings = ['sun']`
 The list of EIA 923 Generation Fuel strings associated with solar power.

Type `list`

`pudl.constants.fuel_type_eia923_gen_fuel_waste_strings = ['ab', 'blq', 'msb', 'msn', 'msw', 'ol']`
 The list of EIA 923 Generation Fuel strings associated with solid waste fuel.

Type `list`

`pudl.constants.fuel_type_eia923_gen_fuel_wind_strings = ['wnd']`
 The list of EIA 923 Generation Fuel strings associated with wind power.

Type `list`

`pudl.constants.fuel_units_eia923 = {'barrels': 'Barrels (for liquids)', 'mcf': 'Thousands of cubic feet', 'mwh': 'Million Watts Hours'}`
 A dictionary mapping EIA 923 fuel units (keys) to fuel unit descriptions (values).

Type `dict`

`pudl.constants.generation_fuel_map_eia923 = pudl.constants.generation_fuel_map_eia923`
 A DataFrame of metadata from EIA 923 Generation Fuel.

Type `pandas.DataFrame`

`pudl.constants.generator_assn_map_eia860 = utility_id_eia ... uprate_derate_completed_year`
 A DataFrame of metadata from EIA 860 Generator.

Type `pandas.DataFrame`

`pudl.constants.generator_map_eia923 = plant_id_eia ... report_year year_index ... 2008 plant_id_eia`
 A DataFrame of metadata from EIA 923 Generators.

Type `pandas.DataFrame`

`pudl.constants.generator_proposed_assn_map_eia860 = utility_id_eia utility_name ... cofire_derate_completed_year`
 A DataFrame of metadata from EIA 860 Generator Proposed.

Type `pandas.DataFrame`

`pudl.constants.generator_retired_assn_map_eia860 = utility_id_eia ... switch_oil_gas_year_index`
 A DataFrame of metadata from EIA 860 Generator Retired.

Type `pandas.DataFrame`

`pudl.constants.glue_pudl_tables = ('plants_eia', 'plants_ferc', 'plants', 'utilities_eia', 'utilities_ferc')`
 A dictionary of dictionaries containing EPA IPM tables (keys) and items for each table to be renamed along with the replacement name (values).

Type `dict`

`pudl.constants.licenses = {'cc-by-4.0': {'name': 'CC-BY-4.0', 'path': 'https://creativecommons.org/licenses/by/4.0/'}}, {'cc-by-nc-4.0': {'name': 'CC-BY-NC-4.0', 'path': 'https://creativecommons.org/licenses/by-nc/4.0/'}}, {'cc-by-nd-4.0': {'name': 'CC-BY-ND-4.0', 'path': 'https://creativecommons.org/licenses/by-nd/4.0/'}}, {'cc-by-sa-4.0': {'name': 'CC-BY-SA-4.0', 'path': 'https://creativecommons.org/licenses/by-sa/4.0/'}}`
 A dictionary of dictionaries containing license types and their attributes.

Type `dict`

`pudl.constants.missing_respondents_ferc1 = {514: 'respondent_514', 515: 'respondent_515'}`
 A dictionary of missing FERC Form 1 respondent IDs (keys) and names (values).

Type `dict`

`pudl.constants.month_dict_eia923 = {1: '_january$', 2: '_february$', 3: '_march$', 4: '_april$', 5: '_may$', 6: '_june$', 7: '_july$', 8: '_august$', 9: '_september$', 10: '_october$', 11: '_november$', 12: '_december$'}`
 A dictionary mapping column numbers (keys) to months (values).

Type `dict`

`pudl.constants.need_fix_inting = {'coalmine_eia923': ('mine_id_msha', 'county_id_fips'), 'plants_eia': ('plant_id_eia', 'county_id_fips'), 'plants_ferc': ('plant_id_ferc', 'county_id_fips'), 'utilities_eia': ('utility_id_eia', 'county_id_fips'), 'utilities_ferc': ('utility_id_ferc', 'county_id_fips')}`
 A dictionary containing tables (keys) and column names (values) containing integer-type columns whose null values need fixing.

Type `dict`

`pudl.constants.nerc_region = {'ASCC': 'Alaska Systems Coordinating Council', 'FRCC': 'Florida Reliability Coordinating Council', 'MRO': 'Midcontinent Reliability Council', 'NERC': 'North American Reliability Council', 'PJM': 'PJM Reliability Council', 'SPP': 'Southwest Power Pool', 'WECC': 'Western Electricity Reliability Council'}`
 A dictionary mapping NERC Region abbreviations (keys) to NERC Region names (values).

Type `dict`

`pudl.constants.output_formats = ['sqlite', 'parquet', 'datapackage', 'notebook']`
 A list of types of PUDL output formats.

Type `list`

`pudl.constants.ownership_assn_map_eia860 = utility_id_eia utility_name ... owner_zip_code`
 A DataFrame of metadata from EIA 860 Ownership.

Type `pandas.DataFrame`

`pudl.constants.plant_assn_map_eia860 = utility_id_eia ... liquefied_natural_gas_storage_year`
 A DataFrame of metadata from EIA 860 Plant.

Type `pandas.DataFrame`

`pudl.constants.plant_frame_map_eia923 = report_year ... nameplate_capacity_mw year_index ...`
 A DataFrame of metadata from EIA 923 Plant Frame.

Type `pandas.DataFrame`

`pudl.constants.prime_movers = ['steam_turbine', 'gas_turbine', 'hydro', 'internal_combustion']`
 A list of the types of prime movers

Type `list`

`pudl.constants.prime_movers_eia923 = {'BA': 'Energy Storage, Battery', 'BT': 'Turbines Used in Base Load', ...}`
 A dictionary mapping EIA 923 prime mover codes (keys) and prime mover names/descriptions (values).

Type `dict`

`pudl.constants.pudl_tables = {'eia860': ('boiler_generator_assn_eia860', 'utilities_eia860'), ...}`
 A dictionary containing data sources (keys) and the list of associated tables from that datasource that can be pulled into PUDL (values).

Type `dict`

`pudl.constants.read_excel_epaipm_dict = {'load_curves_epaipm': {'skiprows': 3, 'usecols': 'A:D'}, ...}`
 A dictionary of dictionaries containing EPA IPM tables and associated information for reading those tables into PUDL (values).

Type `dict`

`pudl.constants.rto_iso = {'CAISO': 'California ISO', 'ERCOT': 'Electric Reliability Council of Texas', ...}`
 A dictionary containing ISO/RTO abbreviations (keys) and names (values)

Type `dict`

`pudl.constants.sector_eia = {'1': 'Electric Utility', '2': 'NAICS-22 Non-Cogen', '3': 'NAICS-23 Electric, Electronic and Other Equipment', ...}`
 A dictionary mapping EIA numeric codes (keys) to EIA's internal consolidated NAICS sectors (values).

Type `dict`

`pudl.constants.skiprows_eia860 = boiler_generator_assn ... generator_retired year_index ...`
 A DataFrame of metadata from EIA 860 skiprows map.

Type `pandas.DataFrame`

`pudl.constants.skiprows_eia923 = generation_fuel_stocks ... fuel_receipts_costs plant_frame ...`
 A DataFrame of metadata from the EIA 923 skiprows map.

Type `pandas.DataFrame`

`pudl.constants.state_tz_approx = {'AB': 'America/Edmonton', 'AK': 'US/Alaska', 'AL': 'US/Central', ...}`
 A dictionary containing US and Canadian state/territory abbreviations (keys) and timezones (values)

Type `dict`

`pudl.constants.stocks_map_eia923 = census_division_and_state ... petcoke_december year_index ...`
 A DataFrame of metadata from EIA 923 Stocks.

Type `pandas.DataFrame`

`pudl.constants.tab_map_eia860 = boiler_generator_assn ... generator_retired year_index ...`
 A DataFrame of metadata from EIA 860 tab map.

Type `pandas.DataFrame`

`pudl.constants.tab_map_eia923 = generation_fuel_stocks ... fuel_receipts_costs plant_frame ...`
 A DataFrame of metadata from the EIA 923 tab map.

Type `pandas.DataFrame`

`pudl.constants.table_map_ferc1_pudl = {'accumulated_depreciation_ferc1': 'f1_accumdepr_pr`
 A dictionary mapping PUDL table names (keys) to the corresponding FERC Form 1 DBF table names.

Type `dict`

`pudl.constants.transport_modes_eia923 = {'GL': 'Great Lakes: Shipments of coal moved to c`
 A dictionary mapping primary and secondary transportation mode codes (keys) to descriptions (values).

Type `dict`

`pudl.constants.travis_ci_eia860_years = (2017,)`
 A tuple containing years of EIA 860 data to use with Travis continuous integration.

Type `tuple`

`pudl.constants.travis_ci_eia923_years = (2017,)`
 A tuple containing years of EIA 923 data to use with Travis continuous integration.

Type `tuple`

`pudl.constants.travis_ci_epacems_states = ('ID',)`
 A tuple containing states whose EPA CEMS data are used with Travis continuous integration.

Type `tuple`

`pudl.constants.travis_ci_epacems_years = (2017,)`
 A tuple containing years of EPA CEMS data to use with Travis continuous integration.

Type `tuple`

`pudl.constants.travis_ci_ferc1_years = (2017,)`
 A tuple containing years of FERC1 data to use with Travis continuous integration.

Type `tuple`

`pudl.constants.us_states = {'AK': 'Alaska', 'AL': 'Alabama', 'AR': 'Arkansas', 'AS': 'Amer`
 A dictionary containing US state abbreviations (keys) and names (values)

Type `dict`

`pudl.constants.utility_assn_map_eia860 = utility_id_eia ... entity_type year_index ... 200`
 A DataFrame of metadata from EIA 860 Utility.

Type `pandas.DataFrame`

`pudl.constants.working_years = {'eia860': (2011, 2012, 2013, 2014, 2015, 2016, 2017), 'eia`
 A dictionary of data sources (keys) and tuples containing the years for each data source that are able to be ingested into PUDL.

Type `dict`

`pudl.constants.xlsx_maps_pkg = 'pudl.package_data.meta.xlsx_maps'`
 type?:

Todo: Return to

pudl.etl module

Module coordinating the PUDL ETL pipeline, generating data packages.

The PUDL project integrates several different public data sets into well normalized data packages allowing easier access and interaction between all each dataset. This module coordinates the extract/transform/load process of data from:

- US Energy Information Agency (EIA): - Form 860 (eia860) - Form 923 (eia923)
- US Federal Energy Regulatory Commission (FERC): - Form 1 (ferc1)
- US Environmental Protection Agency (EPA): - Continuous Emissions Monitory System (epacems) - Integrated Planning Model (epaipm)

`pudl.etl.etl_pkg(pkg_settings, pudl_settings, pkg_bundle_dir)`

Extracts, transforms and loads CSVs.

This is the coordinating function for generating all of the CSV's for a data package. For each of the datasets enumerated in the `pkg_settings`, this function runs the dataset specific ETL function. Along the way, we are accumulating which tables have been loaded. This is useful for generating the metadata associated with the package.

Parameters

- **pkg_settings** (*dict*) – a dictionary of `etl_params` for a datapackage.
- **pudl_settings** (*dict*) – a dictionary filled with settings that mostly describe paths to various resources and outputs.
- **uuid_pkgs** (*uuid*) –

Returns dictionary with datapackpackages (keys) and lists of tables (values)

Return type `dict`

`pudl.etl.generate_data_packages(pkg_bundle_settings, pudl_settings, pkg_bundle_name, debug=False, clobber=False)`

Coordinate the generation of data packages.

For each bundle of packages laid out in the `package_settings`, this function generates data packages. First, the settings are validated (which runs through each of the settings listed in the `package_settings`). Then for each of the packages, run through the `etl` (extract, transform, load) functions, which generates CSVs. Then the metadata for the packages is generated by pulling from the metadata (which is a json file containing the schema for all of the possible pudl tables).

Parameters

- **pkg_bundle_settings** (*iterable*) – a list of dictionaries. Each item in the list corresponds to a data package. Each data package's dictionary contains the arguments for its ETL function.
- **pudl_settings** (*dict*) – a dictionary filled with settings that mostly describe paths to various resources and outputs.
- **pkg_bundle_name** (*string*) – name of directory you want the bundle of data packages to live.
- **debug** (*bool*) – If True, return a dictionary with package names (keys) and a list with the data package metadata and report (values).
- **clobber** (*bool*) –

Returns A tuple containing generated metadata for the packages laid out in the `package_settings`.

Return type `tuple`

`pudl.etl.get_flattened_etl_parameters(pkg_bundle_settings)`

Compile flattened etl parameters.

Parameters `pkg_bundle_settings` (*iterable*) – a list of data package parameters, with each element of the list being a dictionary specifying the data to be packaged.

Returns dictionary of etl parameters (i.e. `ferc1_years`, `eia923_years`)

Return type `dict`

`pudl.etl.validate_params(pkg_bundle_settings, pudl_settings)`

Read and validate the etl inputs from a settings file.

Parameters `pkg_bundle_settings` (*iterable*) – a list of data package parameters, with each element of the list being a dictionary specifying the data to be packaged.

Returns validated list of inputs

Return type `iterable`

pudl.helpers module

General utility functions that are used in a variety of contexts.

The functions in this module are used in various stages of the ETL and post-etl processes. They are usually not dataset specific, but not always. If a function is designed to be used as a general purpose tool, applicable in multiple scenarios, it should probably live here. There are lots of transform type functions in here that help with cleaning and restructuring dataframes.

`pudl.helpers.cleanstrings(df, columns, stringmaps, unmapped=None, simplify=True)`

Consolidate freeform strings in several dataframe columns.

This function will consolidate freeform strings found in `columns` into simplified categories, as defined by `stringmaps`. This is useful when a field contains many different strings that are really meant to represent a finite number of categories, e.g. a type of fuel. It can also be used to create simplified categories that apply to similar attributes that are reported in various data sources from different agencies that use their own taxonomies.

The function takes and returns a `pandas.DataFrame`, making it suitable for use with the `pd.DataFrame.pipe()` method in a chain.

Parameters

- **df** (`pd.DataFrame`) – the `DataFrame` containing the string columns to be cleaned up.
- **columns** (`list`) – a list of string column labels found in the column index of `df`. These are the columns that will be cleaned.
- **stringmaps** (`list`) – a list of dictionaries. The keys of these dictionaries are strings, and the values are lists of strings. Each dictionary in the list corresponds to a column in `columns`. The keys of the dictionaries are the values with which every string in the list of values will be replaced.
- **unmapped** (`str`, `None`) – the value with which strings not found in the stringmap dictionary will be replaced. Typically the null string `''`. If `None`, then strings found in the columns but not in the stringmap will be left unchanged.
- **simplify** (`bool`) – If true, strip whitespace, remove duplicate whitespace, and force lower-case on both the string map and the values found in the columns to be cleaned. This can reduce the overall number of string values that need to be tracked.

Returns The function returns a new pandas series/column that can be used to set the values of the original data.

Return type `pandas.Series`

Todo: Update docstring.

`pudl.helpers.cleanstrings_series(col, str_map, unmapped=None, simplify=True)`

Clean up the strings in a single column/Series.

Parameters

- **col** (*pd.Series*) – A pandas Series, typically a single column of a dataframe, containing the freeform strings that are to be cleaned.
- **map** (*dict*) – A dictionary of lists of strings, in which the keys are the simplified canonical strings, with which each string found in the corresponding list will be replaced.
- **unmapped** (*str*) – A value with which to replace any string found in col that is not found in one of the lists of strings in map. Typically the null string ''. If None, these strings will not be replaced.
- **simplify** (*bool*) – If True, strip and compact whitespace, and lowercase all strings in both the list of values to be replaced, and the values found in col. This can reduce the number of strings that need to be kept track of.

Returns The cleaned up Series / column, suitable for replacing the original messy column in a `pd.DataFrame`.

Return type `pandas.Series`

`pudl.helpers.convert_to_date(df, date_col='report_date', year_col='report_year', month_col='report_month', day_col='report_day', month_value=1, day_value=1)`

Convert specified year, month or day columns into a datetime object.

Parameters

- **df** (*pandas.DataFrame*) – dataframe to convert
- **date_col** (*str*) – the name of the column you want in the output.
- **year_col** (*str*) – the name of the year column in the original table.
- **month_col** (*str*) – the name of the month column in the original table.
- **day_col** – the name of the day column in the original table.
- **month_value** (*int*) – generated month if no month exists.
- **day_value** (*int*) – generated day if no month exists.

Returns A `DataFrame` in which the year, month, day columns values have been converted into datetime objects.

Return type `pandas.DataFrame`

Todo: Update docstring.

`pudl.helpers.drop_tables(engine)`

Drops all tables from a SQLite database.

Creates an `sa.schema.MetaData` object reflecting the structure of the database that the passed in `engine` refers to, and uses that schema to drop all existing tables.

Todo: Treat DB connection as a context manager (with/as).

Parameters `engine` (*sa.engine.Engine*) – An SQL Alchemy SQLite database Engine pointing at an existing SQLite database to be deleted.

Returns None

`pudl.helpers.extend_annual(df, date_col='report_date', start_date=None, end_date=None)`

Extend time range in a DataFrame by duplicating first and last years.

Takes the earliest year's worth of annual data and uses it to create earlier years by duplicating it, and changing the year. Similarly, extends a dataset into the future by duplicating the last year's records.

This is primarily used to extend the EIA860 data about utilities, plants, and generators, so that we can analyze a larger set of EIA923 data. EIA923 data has been integrated a bit further back, and the EIA860 data has a year long lag in being released.

Parameters

- `df` (*pandas.DataFrame*) –
- `date_col` (*str*) –
- `start_date` (*date*) –
- `end_date` (*date*) –

Returns

Return type *pandas.DataFrame*

Todo: Return to

`pudl.helpers.find_timezone(*, lng=None, lat=None, state=None, strict=True)`

Find the timezone associated with the a specified input location.

Note that this function requires named arguments. The names are `lng`, `lat`, and `state`. `lng` and `lat` must be provided, but they may be NA. `state` isn't required, and isn't used unless `lng/lat` are NA or `timezonefinder` can't find a corresponding timezone.

Timezones based on states are imprecise, so it's far better to use `lng/lat` if possible. If `strict` is True, `state` will not be used. More on state-to-timezone conversion here: https://en.wikipedia.org/wiki/List_of_time_offsets_by_U.S._state_and_territory

Parameters

- `lng` (*int or float in [-180, 180]*) – Longitude, in decimal degrees
- `lat` (*int or float in [-90, 90]*) – Latitude, in decimal degrees
- `state` (*str*) – Abbreviation for US state or Canadian province
- `strict` (*bool*) – Raise an error if no timezone is found?

Returns The timezone (as an IANA string) for that location.

Return type *str*

Todo: Update docstring.

`pudl.helpers.fix_eia_na(df)`

Replace common ill-posed EIA NA spreadsheet values with `np.nan`.

Parameters `df` (`pandas.DataFrame`) – The DataFrame to clean.

Returns The cleaned DataFrame.

Return type `pandas.DataFrame`

Todo: Update docstring.

`pudl.helpers.fix_int_na(df, columns, float_na=nan, int_na=-1, str_na="")`

Convert NA containing integer columns from float to string.

Numpy doesn't have a real NA value for integers. When pandas stores integer data which has NA values, it thus upcasts integers to floating point values, using `np.nan` values for NA. However, in order to dump some of our dataframes to CSV files that are suitable for loading into postgres directly, we need to write out integer formatted numbers, with empty strings as the NA value. This function replaces `np.nan` values with a sentinel value, converts the column to integers, and then to strings, finally replacing the sentinel value with the desired NA string.

Parameters

- `df` (`pandas.DataFrame`) – The dataframe to be fixed. This argument allows method chaining with the `pipe()` method.
- `columns` (*iterable of strings*) – A list of DataFrame column labels indicating which columns need to be reformatted for output.
- `float_na` (`float`) – The floating point value to be interpreted as NA and replaced in col.
- `int_na` (`int`) – Sentinel value to substitute for `float_na` prior to conversion of the column to integers.
- `str_na` (`str`) – `sa.String` value to substitute for `int_na` after the column has been converted to strings.

Returns a new DataFrame, with the selected columns converted to strings that look like integers, compatible with the postgresql COPY FROM command.

Return type `df` (`pandas.DataFrame`)

Todo: Update docstring.

`pudl.helpers.is_annual(df_year, year_col='report_date')`

Determine whether dataframe is consistent with yearly reporting.

Some processes will only work with consistent yearly reporting. This means if you have two non-contiguous years of data or the datetime reporting is inconsistent, the process will break.

Parameters

- `()` (`df_year`) –
- `year_col` (`str`) – The column of the DataFrame in which the year is reported.

Returns

Return type `bool`

Todo: Return to for df_year, Returns, assert statements

`pudl.helpers.merge_dicts(list_of_dicts)`

Merge multiple dictionaries together.

Given any number of dicts, shallow copy and merge into a new dict, precedence goes to key value pairs in latter dicts. :param dict_args: a list of dictionaries. :type dict_args: list

Returns dictionary

`pudl.helpers.merge_on_date_year(df_date, df_year, on=(), how='inner', date_col='report_date', year_col='report_date')`

Merge two dataframes based on a shared year.

Some of our data is annual, and has an integer year column (e.g. FERC 1). Some of our data is annual, and uses a Date column (e.g. EIA 860), and some of our data has other temporal resolutions, and uses date columns (e.g. EIA 923 fuel receipts are monthly, EPA CEMS data is hourly). This function takes two data frames and merges them based on the year that the data pertains to. It requires one of the dataframes to have annual resolution, and allows the annual time to be described as either an integer year or a Date. The non-annual dataframe must have a Date column.

By default, it is assumed that both the date and annual columns to be merged on are called 'report_date' since that's the common case when bringing together EIA860 and EIA923 data.

Parameters

- **df_date** – the dataframe with a more granular date column, the label of which is specified by date_col (report_date by default)
- **df_year** – the dataframe with a column containing annual dates, the label of which is specified by year_col (report_date by default)
- **on** – The list of columns to merge on, other than the year and date columns.
- **date_col** – name of the date column to use to find the year to merge on. Must be a Date.
- **year_col** – name of the year column to merge on. Must be a Date column with annual resolution.

Returns a dataframe with a date column, but no year columns, and only one copy of any shared columns that were not part of the list of columns to be merged on. The values from df1 are the ones which are retained for any shared, non-merging columns

Return type `pandas.DataFrame`

`pudl.helpers.month_year_to_date(df)`

Convert all pairs of year/month fields in a dataframe into Date fields.

This function finds all column names within a dataframe that match the regular expression '_month\$' and '_year\$', and looks for pairs that have identical prefixes before the underscore. These fields are assumed to describe a date, accurate to the month. The two fields are used to construct a new _date column (having the same prefix) and the month/year columns are then dropped.

This function needs to be combined with convert_to_date, and improved:

- find and use a _day\$ column as well
- allow specification of default month & day values, if none are found.
- allow specification of lists of year, month, and day columns to be combined, rather than automatically finding all the matching ones.
- Do the Right Thing when invalid or NA values are encountered.

Parameters `df` (*pandas.DataFrame*) – The DataFrame in which to convert year/months fields to Date fields.

Returns A DataFrame in which the year/month fields have been converted into Date fields.

Return type *pandas.DataFrame*

Todo: Update docstring.

`pudl.helpers.organize_cols(df, cols)`

Organize columns into key ID & name fields & alphabetical data columns.

For readability, it's nice to group a few key columns at the beginning of the dataframe (e.g. `report_year` or `report_date`, `plant_id`...) and then put all the rest of the data columns in alphabetical order.

Parameters

- `df` – The DataFrame to be re-organized.
- `cols` – The columns to put first, in their desired output ordering.

Returns A dataframe with the same columns as the input DataFrame `df`, but with `cols` first, in the same order as they were passed in, and the remaining columns sorted alphabetically.

Return type *pandas.DataFrame*

Todo: Update docstring.

`pudl.helpers.simplify_columns(df)`

Simplify column labels for use as database fields.

This transformation includes:

- Replacing all non-alphanumeric characters with spaces.
- Forcing all letters to be lower case.
- Compacting internal whitespace.
- Stripping leading and trailing whitespace.

Parameters `df` (*pandas.DataFrame*) – The DataFrame to clean.

Returns The cleaned DataFrame.

Return type *pandas.DataFrame*

Todo: Update docstring.

`pudl.helpers.strip_lower(df, columns=None)`

Strip and compact whitespace, lowercase listed DataFrame columns.

First converts all listed columns (if present in `df`) to string type, then applies the `str.strip()` and `str.lower()` methods to them, and replaces all internal whitespace to a single space. The columns are assigned in place.

Parameters

- `df` (*pandas.DataFrame*) – DataFrame whose columns are being cleaned up.

- **columns** (*iterable*) – The labels of the columns to be stripped and converted to lower-case.

Returns The whole DataFrame that was passed in, with the columns cleaned up in place, allowing method chaining.

Return type `pandas.DataFrame`

```
pudl.helpers.sum_na(self, axis=None, *, skipna=False, level=None, numeric_only=None,
                    min_count=0, **kwargs)
```

Return the sum of the values for the requested axis.

This is equivalent to the method `numpy.sum`.

Parameters

- **axis** (*{index (0)}*) – Axis for the function to be applied on.
- **skipna** (*bool, default True*) – Exclude NA/null values when computing the result.
- **level** (*int or level name, default None*) – If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a scalar.
- **numeric_only** (*bool, default None*) – Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.
- **min_count** (*int, default 0*) – The required number of valid values to perform the operation. If fewer than `min_count` non-NA values are present the result will be NA.

New in version 0.22.0: Added with the default being 0. This means the sum of an all-NA or empty Series is 0, and the product of an all-NA or empty Series is 1.

- ****kwargs** – Additional keyword arguments to be passed to the function.

Returns

Return type scalar or Series (if level specified)

See also:

Series.sum() Return the sum.

Series.min() Return the minimum.

Series.max() Return the maximum.

Series.idxmin() Return the index of the minimum.

Series.idxmax() Return the index of the maximum.

DataFrame.sum() Return the sum over the requested axis.

DataFrame.min() Return the minimum over the requested axis.

DataFrame.max() Return the maximum over the requested axis.

DataFrame.idxmin() Return the index of the minimum over the requested axis.

DataFrame.idxmax() Return the index of the maximum over the requested axis.

Examples

```
>>> idx = pd.MultiIndex.from_arrays([
...     ['warm', 'warm', 'cold', 'cold'],
...     ['dog', 'falcon', 'fish', 'spider']],
...     names=['blooded', 'animal'])
>>> s = pd.Series([4, 2, 0, 8], name='legs', index=idx)
>>> s
blooded  animal
warm      dog      4
          falcon   2
cold      fish     0
          spider   8
Name: legs, dtype: int64
```

```
>>> s.sum()
14
```

Sum using level names, as well as indices.

```
>>> s.sum(level='blooded')
blooded
warm      6
cold      8
Name: legs, dtype: int64
```

```
>>> s.sum(level=0)
blooded
warm      6
cold      8
Name: legs, dtype: int64
```

By default, the sum of an empty or all-NA Series is 0.

```
>>> pd.Series([]).sum() # min_count=0 is the default
0.0
```

This can be controlled with the `min_count` parameter. For example, if you'd like the sum of an empty series to be NaN, pass `min_count=1`.

```
>>> pd.Series([]).sum(min_count=1)
nan
```

Thanks to the `skipna` parameter, `min_count` handles all-NA and empty series identically.

```
>>> pd.Series([np.nan]).sum()
0.0
```

```
>>> pd.Series([np.nan]).sum(min_count=1)
nan
```

`pudl.helpers.verify_input_files(ferc1_years, eia923_years, eia860_years, epacems_years, epacems_states, pudl_settings)`

Verify that all required data files exist prior to the ETL process.

Parameters

- **ferc1_years** (*iterable*) – Years of FERC1 data we're going to import.

- **eia923_years** (*iterable*) – Years of EIA923 data we’re going to import.
- **eia860_years** (*iterable*) – Years of EIA860 data we’re going to import.
- **epacems_years** (*iterable*) – Years of CEMS data we’re going to import.
- **epacems_states** (*iterable*) – States of CEMS data we’re going to import.
- **data_dir** (*path-like*) – Path to the top level of the PUDL datastore.

Raises **FileNotFoundError** – If any of the requested data is missing.

Todo: Check Docstring.

pudl.validate module

PUDL data validation functions and test case specifications.

What defines a data validation?

- What data are we checking? * What table or output does it come from? * What selection criteria do we apply to that table or output?
- What are we checking it against? * Itself (helps validate that the tests themselves are working) * A processed version of itself (aggregation or derived values) * A hard-coded external standard (e.g. heat rates, fuel heat content)

```
pudl.validate.bf_eia923_agg = [{'title': 'Coal ash content', 'query': "fuel_type_code_pudl = 'EIA923 Boiler Fuel'"}]
```

EIA923 Boiler Fuel data validation against aggregated data.

```
pudl.validate.bf_eia923_coal_ash_content = [{'title': 'Bituminous coal ash content (middle)', 'query': "fuel_type_code_pudl = 'EIA923 Boiler Fuel'"}]
```

Valid coal ash content (%). Based on historical reporting in EIA 923.

```
pudl.validate.bf_eia923_coal_heat_content = [{'title': 'Bituminous coal heat content (middle)', 'query': "fuel_type_code_pudl = 'EIA923 Boiler Fuel'"}]
```

Valid coal (bituminous, sub-bituminous, and lignite) heat content values.

Based on IEA coal grade definitions: <https://www.iea.org/statistics/resources/balanceddefinitions/>

```
pudl.validate.bf_eia923_coal_sulfur_content = [{'title': 'Coal sulfur content (tails)', 'query': "fuel_type_code_pudl = 'EIA923 Boiler Fuel'"}]
```

Valid coal sulfur content values.

Based on historically reported values in EIA 923 Fuel Receipts and Costs.

```
pudl.validate.bf_eia923_gas_heat_content = [{'title': 'Natural Gas heat content (middle)', 'query': "fuel_type_code_pudl = 'EIA923 Boiler Fuel'"}]
```

Valid natural gas heat content values.

Based on historically reported values in EIA 923 Fuel Receipts and Costs. May fail because of a population of bad data around 0.1 mmbtu/unit. This appears to be an off-by-10x error, possibly due to reporting error in units used.

```
pudl.validate.bf_eia923_oil_heat_content = [{'title': 'Diesel Fuel Oil heat content (tails)', 'query': "fuel_type_code_pudl = 'EIA923 Boiler Fuel'"}]
```

Valid petroleum based fuel heat content values.

Based on historically reported values in EIA 923 Fuel Receipts and Costs.

```
pudl.validate.bf_eia923_self = [{'title': 'Bituminous coal ash content', 'query': "fuel_type_code_pudl = 'EIA923 Boiler Fuel'"}]
```

EIA923 Boiler Fuel data validation against itself.

```
pudl.validate.bounds_histogram(df, data_col, weight_col, query, low_q, hi_q, low_bound, hi_bound, title="")
```

Plot a weighted histogram showing acceptable bounds/actual values.

`pudl.validate.frc_eia923_agg = [{'title': 'Coal ash content', 'query': "fuel_type_code_pudl"}]`
 EIA923 fuel receipts & costs data validation against aggregated data.

`pudl.validate.frc_eia923_coal_ash_content = [{'title': 'Bituminous coal ash content (middle)'}]`
 Valid coal ash content (%). Based on historical reporting in EIA 923.

`pudl.validate.frc_eia923_coal_heat_content = [{'title': 'Bituminous coal heat content (middle)'}]`
 Valid coal (bituminous, sub-bituminous, and lignite) heat content values.
 Based on IEA coal grade definitions: <https://www.iea.org/statistics/resources/balanceddefinitions/>

`pudl.validate.frc_eia923_coal_mercury_content = [{'title': 'Coal mercury content (upper tail)'}]`
 Valid coal mercury content limits.
 Based on USGS FS095-01: <https://pubs.usgs.gov/fs/fs095-01/fs095-01.html> Upper tail may fail because of a population of extremely high mercury content coal (9.0ppm) which is likely a reporting error.

`pudl.validate.frc_eia923_coal_moisture_content = [{'title': 'Bituminous coal moisture content (middle)'}]`
 Valid coal moisture content, based on historical EIA 923 reporting.

`pudl.validate.frc_eia923_coal_sulfur_content = [{'title': 'Coal sulfur content (tails)'}]`
 Valid coal sulfur content values.
 Based on historically reported values in EIA 923 Fuel Receipts and Costs.

`pudl.validate.frc_eia923_gas_heat_content = [{'title': 'Natural Gas heat content (middle)'}]`
 Valid natural gas heat content values.
 Based on historically reported values in EIA 923 Fuel Receipts and Costs. May fail because of a population of bad data around 0.1 mmbtu/unit. This appears to be an off-by-10x error, possibly due to reporting error in units used.

`pudl.validate.frc_eia923_oil_heat_content = [{'title': 'Diesel Fuel Oil heat content (tail)'}]`
 Valid petroleum based fuel heat content values.
 Based on historically reported values in EIA 923 Fuel Receipts and Costs.

`pudl.validate.frc_eia923_self = [{'title': 'Bituminous coal ash content', 'query': "energy_type_code_pudl"}]`
 EIA923 fuel receipts & costs data validation against itself.

`pudl.validate.gf_eia923_agg = [{'title': 'Coal heat content', 'query': "fuel_type_code_pudl"}]`
 EIA923 Boiler Fuel data validation against aggregated data.

`pudl.validate.gf_eia923_coal_heat_content = [{'title': 'All coal heat content (middle)'}]`
 Valid coal heat content values (all coal types).
 The Generation Fuel table does not break different coal types out separately, so we can only test the validity of the entire suite of coal records.
 Based on IEA coal grade definitions: <https://www.iea.org/statistics/resources/balanceddefinitions/>

`pudl.validate.gf_eia923_gas_heat_content = [{'title': 'Natural Gas heat content (middle)'}]`
 Valid natural gas heat content values.
 Based on historically reported values in EIA 923 Fuel Receipts and Costs. May fail because of a population of bad data around 0.1 mmbtu/unit. This appears to be an off-by-10x error, possibly due to reporting error in units used.

`pudl.validate.gf_eia923_oil_heat_content = [{'title': 'Diesel Fuel Oil heat content (tail)'}]`
 Valid petroleum based fuel heat content values.
 Based on historically reported values in EIA 923 Fuel Receipts and Costs.

`pudl.validate.historical_distribution(df, data_col, weight_col, quantile)`

Calculate a historical distribution of weighted values of a column.

In order to know what a “reasonable” value of a particular column is in the pudl data, we can use this function to see what the value in that column has been in each of the years of data we have on hand, and a given quantile. This population of values can then be used to set boundaries on acceptable data distributions in the aggregated and processed data.

Parameters

- **df** (*pandas.DataFrame*) – a dataframe containing historical data, with a column named either `report_date` or `report_year`.
- **data_col** (*string*) – Label of the column containing the data of interest.
- **weight_col** (*string*) – Label of the column containing the weights to be used in scaling the data.

Returns The weighted quantiles of data, for each of the years found in the historical data of df.

Return type [list](#)

`pudl.validate.historical_histogram(orig_df, test_df, data_col, weight_col, query="", low_q=0.05, mid_q=0.5, hi_q=0.95, low_bound=None, hi_bound=None, title=")`

Weighted histogram comparing distribution with historical subsamples.

`pudl.validate.mcoe_coal_capacity_factor = [{'title': 'Coal Capacity Factor (middle)', 'query': 'mcoe_coal_capacity_factor'}]`
Static constraints on coal fired generator capacity factors.

`pudl.validate.mcoe_coal_heat_rate = [{'title': 'Coal Unit Heat Rates (middle)', 'query': 'mcoe_coal_heat_rate'}]`
Static constraints on coal fired generator heat rates.

`pudl.validate.mcoe_fuel_cost_per_mmbtu = [{'title': 'Coal Fuel Costs (middle)', 'query': 'mcoe_fuel_cost_per_mmbtu'}]`
Static constraints on fuel costs per mmbtu of fuel consumed.

`pudl.validate.mcoe_fuel_cost_per_mwh = [{'title': 'Coal Fuel Costs (middle)', 'query': 'mcoe_fuel_cost_per_mwh'}]`
Static constraints on fuel costs per MWh net generation.

`pudl.validate.mcoe_gas_capacity_factor = [{'title': 'Natural Gas Capacity Factor (middle)', 'query': 'mcoe_gas_capacity_factor'}]`
Static constraints on natural gas generator capacity factors.

`pudl.validate.mcoe_gas_heat_rate = [{'title': 'Natural Gas Unit Heat Rates (middle, 2015+)', 'query': 'mcoe_gas_heat_rate'}]`
Static constraints on gas fired generator heat rates.

`pudl.validate.plot_vs_agg(orig_df, agg_df, validation_cases)`
Validate a bunch of distributions against aggregated versions.

`pudl.validate.plot_vs_bounds(df, validation_cases)`
Run through a data validation based on absolute bounds.

`pudl.validate.plot_vs_self(df, validation_cases)`
Validate a bunch of distributions against themselves.

`pudl.validate.vs_bounds(df, data_col, weight_col, query="", title="", low_q=False, low_bound=False, hi_q=False, hi_bound=False)`
Test a distribution against an upper bound, lower bound, or both.

`pudl.validate.vs_historical(orig_df, test_df, data_col, weight_col, query="", low_q=0.05, mid_q=0.5, hi_q=0.95, title=")`
Validate aggregated distributions against original data.

`pudl.validate.vs_self(df, data_col, weight_col, query="", title="", low_q=0.05, mid_q=0.5, hi_q=0.95)`
Test a distribution against its own historical range.

This is a special case of the `pudl.validate.vs_historical` function, in which both the `orig_df` and `test_df` are the same. Mostly it helps ensure that the test itself is valid for the given distribution.

`pudl.validate.weighted_quantile(data, weights, quantile)`

Calculate the weighted quantile of a Series or DataFrame column.

This function allows us to take two columns from a `pandas.DataFrame` one of which contains an observed value (data) like heat content per unit of fuel, and the other of which (weights) contains a quantity like quantity of fuel delivered which should be used to scale the importance of the observed value in an overall distribution, and calculate the values that the scaled distribution will have at various quantiles.

Parameters

- **data** (`pandas.Series`) – A series containing numeric data.
- **weights** (`pandas.series`) – Weights to use in scaling the data. Must have the same length as data.
- **quantile** (`float`) – A number between 0 and 1, representing the quantile at which we want to find the value of the weighted data.

Returns the value in the weighted data corresponding to the given quantile. If there are no values in the data, return `numpy.na`.

Return type `float`

Module contents

The Public Utility Data Liberation (PUDL) Project.

PYTHON MODULE INDEX

p

- `pudl`, 132
- `pudl.analysis`, 49
- `pudl.analysis.mcoe`, 48
- `pudl.cli`, 106
- `pudl.constants`, 107
- `pudl.convert`, 53
 - `pudl.convert.datapkg_to_sqlite`, 49
 - `pudl.convert.epacems_to_parquet`, 50
 - `pudl.convert.ferc1_to_sqlite`, 51
 - `pudl.convert.flatten_datapkg`, 51
- `pudl.etl`, 119
- `pudl.extract`, 67
 - `pudl.extract.eia860`, 53
 - `pudl.extract.eia923`, 55
 - `pudl.extract.epacems`, 57
 - `pudl.extract.epaipm`, 58
 - `pudl.extract.ferc1`, 59
- `pudl.glue`, 69
 - `pudl.glue.ferc1_eia`, 68
- `pudl.helpers`, 121
- `pudl.load`, 75
 - `pudl.load.csv`, 69
 - `pudl.load.metadata`, 71
- `pudl.output`, 86
 - `pudl.output.eia860`, 75
 - `pudl.output.eia923`, 77
 - `pudl.output.export`, 80
 - `pudl.output.ferc1`, 80
 - `pudl.output.glue`, 82
 - `pudl.output.pudltabl`, 82
- `pudl.transform`, 99
 - `pudl.transform.eia`, 86
 - `pudl.transform.eia860`, 87
 - `pudl.transform.eia923`, 89
 - `pudl.transform.epacems`, 91
 - `pudl.transform.epaipm`, 92
 - `pudl.transform.ferc1`, 93
- `pudl.validate`, 129
- `pudl.workspace`, 106
 - `pudl.workspace.datastore`, 99
 - `pudl.workspace.datastore_cli`, 103
 - `pudl.workspace.setup`, 104
 - `pudl.workspace.setup_cli`, 105

A

accumulated_depreciation() (in module *pudl.extract.fercI*), 60
 accumulated_depreciation() (in module *pudl.transform.fercI*), 94
 add() (*pudl.load.csv.BulkCopy* method), 70
 add_facility_id_unit_id_epa() (in module *pudl.transform.epacems*), 91
 add_sqlite_table() (in module *pudl.extract.fercI*), 60
 aer_coal_strings (in module *pudl.constants*), 107
 aer_fuel_type_strings (in module *pudl.constants*), 107
 aer_gas_strings (in module *pudl.constants*), 107
 aer_hydro_strings (in module *pudl.constants*), 107
 aer_nuclear_strings (in module *pudl.constants*), 107
 aer_oil_strings (in module *pudl.constants*), 107
 aer_other_strings (in module *pudl.constants*), 107
 aer_solar_strings (in module *pudl.constants*), 107
 aer_waste_strings (in module *pudl.constants*), 107
 aer_wind_strings (in module *pudl.constants*), 107
 annotated_xlsx() (in module *pudl.output.export*), 80
 assert_valid_param() (in module *pudl.workspace.datastore*), 99

B

base_data_urls (in module *pudl.constants*), 107
 bf_eia923() (*pudl.output.pudltabl.PudlTabl* method), 82
 bf_eia923_agg (in module *pudl.validate*), 129
 bf_eia923_coal_ash_content (in module *pudl.validate*), 129
 bf_eia923_coal_heat_content (in module *pudl.validate*), 129
 bf_eia923_coal_sulfur_content (in module *pudl.validate*), 129
 bf_eia923_gas_heat_content (in module *pudl.validate*), 129
 bf_eia923_oil_heat_content (in module *pudl.validate*), 129

bf_eia923_self (in module *pudl.validate*), 129
 bga() (*pudl.output.pudltabl.PudlTabl* method), 82
 bga_eia860() (*pudl.output.pudltabl.PudlTabl* method), 83
 boiler_fuel() (in module *pudl.transform.eia923*), 89
 boiler_fuel_eia923() (in module *pudl.output.eia923*), 77
 boiler_fuel_map_eia923 (in module *pudl.constants*), 108
 boiler_generator_assn() (in module *pudl.output.glue*), 82
 boiler_generator_assn() (in module *pudl.transform.eia860*), 87
 boiler_generator_assn_eia860() (in module *pudl.output.eia860*), 75
 boiler_generator_assn_map_eia860 (in module *pudl.constants*), 108
 bounds_histogram() (in module *pudl.validate*), 129
 BulkCopy (class in *pudl.load.csv*), 69

C

canada_prov_terr (in module *pudl.constants*), 108
 capacity_factor() (in module *pudl.analysis.mcoe*), 48
 capacity_factor() (*pudl.output.pudltabl.PudlTabl* method), 83
 cems_states (in module *pudl.constants*), 108
 census_region (in module *pudl.constants*), 108
 check_for_matching_parameters() (in module *pudl.convert.flatten_datapkgs*), 52
 check_if_need_update() (in module *pudl.workspace.datastore*), 100
 clean_columns_dump() (in module *pudl.load.csv*), 70
 cleanstrings() (in module *pudl.helpers*), 121
 cleanstrings_series() (in module *pudl.helpers*), 122
 close() (*pudl.load.csv.BulkCopy* method), 70
 coalmine() (in module *pudl.transform.eia923*), 89
 coalmine_country_eia923 (in module *pudl.constants*), 108

coalmine_type_eia923 (in module *pudl.constants*), 108
 compile_data_packages_metadata() (in module *pudl.convert.flatten_datapkgs*), 52
 compile_partitions() (in module *pudl.load.metadata*), 71
 contract_type_eia923 (in module *pudl.constants*), 108
 contributors (in module *pudl.constants*), 108
 contributors_by_source (in module *pudl.constants*), 108
 convert_to_date() (in module *pudl.helpers*), 122
 correct_gross_load_mw() (in module *pudl.transform.epacems*), 91
 cpi_diesel_strings (in module *pudl.constants*), 108
 cpi_geothermal_strings (in module *pudl.constants*), 108
 cpi_natural_gas_strings (in module *pudl.constants*), 108
 cpi_nuclear_strings (in module *pudl.constants*), 108
 cpi_other_strings (in module *pudl.constants*), 109
 cpi_plant_kind_strings (in module *pudl.constants*), 109
 cpi_solar_strings (in module *pudl.constants*), 109
 cpi_steam_strings (in module *pudl.constants*), 109
 cpi_wind_strings (in module *pudl.constants*), 109
 create_cems_schema() (in module *pudl.convert.epacems_to_parquet*), 50
 create_dfs_epaipm() (in module *pudl.extract.epaipm*), 58
 csv_dump() (in module *pudl.load.csv*), 70

D

data_source_info (in module *pudl.constants*), 109
 data_sources (in module *pudl.constants*), 109
 data_sources_from_tables_pkg() (in module *pudl.load.metadata*), 72
 data_years (in module *pudl.constants*), 109
 dbc_filename() (in module *pudl.extract.ferc1*), 60
 dbf2sqlite() (in module *pudl.extract.ferc1*), 61
 dbf_ttypemap (in module *pudl.constants*), 109
 define_sqlite_db() (in module *pudl.extract.ferc1*), 61
 deploy() (in module *pudl.workspace.setup*), 104
 derive_paths() (in module *pudl.workspace.setup*), 104
 dict_dump() (in module *pudl.load.csv*), 71
 download() (in module *pudl.workspace.datastore*), 100
 drop_tables() (in module *pudl.extract.ferc1*), 63
 drop_tables() (in module *pudl.helpers*), 122

E

eia860_pudl_tables (in module *pudl.constants*), 109
 eia923_pudl_tables (in module *pudl.constants*), 109
 energy_source_eia923 (in module *pudl.constants*), 109
 energy_source_eia_simple_map (in module *pudl.constants*), 110
 entities (in module *pudl.constants*), 110
 entity_tables (in module *pudl.constants*), 110
 epacems_additional_plant_info_file (in module *pudl.constants*), 110
 epacems_columns_fill_na_dict (in module *pudl.constants*), 110
 epacems_columns_to_ignore (in module *pudl.constants*), 110
 epacems_csv_dtypes (in module *pudl.constants*), 110
 epacems_rename_dict (in module *pudl.constants*), 110
 epacems_tables (in module *pudl.constants*), 110
 epacems_to_parquet() (in module *pudl.convert.epacems_to_parquet*), 50
 epaipm_pudl_tables (in module *pudl.constants*), 110
 epaipm_region_aggregations (in module *pudl.constants*), 110
 epaipm_region_names (in module *pudl.constants*), 110
 epaipm_url_ext (in module *pudl.constants*), 111
 etl_pkg() (in module *pudl.etl*), 120
 extend_annual() (in module *pudl.helpers*), 123
 extract() (in module *pudl.extract.eia860*), 53
 extract() (in module *pudl.extract.eia923*), 55
 extract() (in module *pudl.extract.epacems*), 57
 extract() (in module *pudl.extract.epaipm*), 58
 extract() (in module *pudl.extract.ferc1*), 63

F

fbp_ferc1() (*pudl.output.pudltabl.PudlTabl* method), 83
 ferc1_lkgal_strings (in module *pudl.constants*), 111
 ferc1_bbl_strings (in module *pudl.constants*), 111
 ferc1_coal_strings (in module *pudl.constants*), 111
 ferc1_const_type_conventional (in module *pudl.constants*), 111
 ferc1_const_type_outdoor (in module *pudl.constants*), 111
 ferc1_const_type_strings (in module *pudl.constants*), 111
 ferc1_data_tables (in module *pudl.constants*), 111

ferc1_dbf2tbl (in module *pudl.constants*), 111
 ferc1_default_tables (in module *pudl.constants*), 111
 ferc1_fuel_strings (in module *pudl.constants*), 111
 ferc1_fuel_unit_strings (in module *pudl.constants*), 111
 ferc1_gal_strings (in module *pudl.constants*), 111
 ferc1_gas_strings (in module *pudl.constants*), 112
 ferc1_gramsU_strings (in module *pudl.constants*), 112
 ferc1_huge_tables (in module *pudl.constants*), 112
 ferc1_kgU_strings (in module *pudl.constants*), 112
 ferc1_mcf_strings (in module *pudl.constants*), 112
 ferc1_mmbtu_strings (in module *pudl.constants*), 112
 ferc1_mwdth_strings (in module *pudl.constants*), 112
 ferc1_mwhth_strings (in module *pudl.constants*), 112
 ferc1_nuke_strings (in module *pudl.constants*), 112
 ferc1_oil_strings (in module *pudl.constants*), 112
 ferc1_other_strings (in module *pudl.constants*), 112
 ferc1_plant_kind_combined_cycle (in module *pudl.constants*), 112
 ferc1_plant_kind_combustion_turbine (in module *pudl.constants*), 112
 ferc1_plant_kind_geothermal (in module *pudl.constants*), 113
 ferc1_plant_kind_nuke (in module *pudl.constants*), 113
 ferc1_plant_kind_photovoltaic (in module *pudl.constants*), 113
 ferc1_plant_kind_solar_thermal (in module *pudl.constants*), 113
 ferc1_plant_kind_steam_turbine (in module *pudl.constants*), 113
 ferc1_plant_kind_strings (in module *pudl.constants*), 113
 ferc1_plant_kind_wind (in module *pudl.constants*), 113
 ferc1_power_purchase_type (in module *pudl.constants*), 113
 ferc1_pudl_tables (in module *pudl.constants*), 113
 ferc1_tbl2dbf (in module *pudl.constants*), 113
 ferc1_ton_strings (in module *pudl.constants*), 113
 ferc1_waste_strings (in module *pudl.constants*), 113
 FERCIFieldParser (class in *pudl.extract.fercI*), 59
 ferc_1_plant_kind_internal_combustion (in module *pudl.constants*), 113
 ferc_accumulated_depreciation (in module *pudl.constants*), 113
 ferc_electric_plant_accounts (in module *pudl.constants*), 114
 FERCPantClassifier (class in *pudl.transform.fercI*), 93
 file_pages_eia860 (in module *pudl.constants*), 114
 files_dict_eia860 (in module *pudl.constants*), 114
 files_dict_epaipm (in module *pudl.constants*), 114
 files_eia860 (in module *pudl.constants*), 114
 find_timezone () (in module *pudl.helpers*), 123
 fit () (*pudl.transform.fercI.FERCPantClassifier* method), 93
 fix_eia_na () (in module *pudl.helpers*), 123
 fix_int_na () (in module *pudl.helpers*), 124
 fix_up_dates () (in module *pudl.transform.epacems*), 91
 flatten_data_package_metadata () (in module *pudl.convert.flatten_datapkg*s), 52
 flatten_data_packages_csvs () (in module *pudl.convert.flatten_datapkg*s), 52
 flatten_pudl_datapackages () (in module *pudl.convert.flatten_datapkg*s), 53
 frc_eia923 () (*pudl.output.pudltabl.PudlTabl* method), 83
 frc_eia923_agg (in module *pudl.validate*), 129
 frc_eia923_coal_ash_content (in module *pudl.validate*), 130
 frc_eia923_coal_heat_content (in module *pudl.validate*), 130
 frc_eia923_coal_mercury_content (in module *pudl.validate*), 130
 frc_eia923_coal_moisture_content (in module *pudl.validate*), 130
 frc_eia923_coal_sulfur_content (in module *pudl.validate*), 130
 frc_eia923_gas_heat_content (in module *pudl.validate*), 130
 frc_eia923_oil_heat_content (in module *pudl.validate*), 130
 frc_eia923_self (in module *pudl.validate*), 130
 fuel () (in module *pudl.extract.fercI*), 64
 fuel () (in module *pudl.transform.fercI*), 95
 fuel_by_plant_ferc1 () (in module *pudl.output.fercI*), 80
 fuel_by_plant_ferc1 () (in module *pudl.transform.fercI*), 95
 fuel_cost () (in module *pudl.analysis.mcoe*), 48
 fuel_cost () (*pudl.output.pudltabl.PudlTabl* method), 83
 fuel_ferc1 () (in module *pudl.output.fercI*), 81
 fuel_ferc1 () (*pudl.output.pudltabl.PudlTabl* method), 83
 fuel_group_eia923 (in module *pudl.constants*), 114
 fuel_group_eia923_simple_map (in module

[pudl.constants](#)), 114
[fuel_receipts_costs\(\)](#) (in module [pudl.transform.eia923](#)), 89
[fuel_receipts_costs_eia923\(\)](#) (in module [pudl.output.eia923](#)), 78
[fuel_receipts_costs_map_eia923](#) (in module [pudl.constants](#)), 114
[fuel_type_aer_eia923](#) (in module [pudl.constants](#)), 114
[fuel_type_eia860_coal_strings](#) (in module [pudl.constants](#)), 114
[fuel_type_eia860_gas_strings](#) (in module [pudl.constants](#)), 114
[fuel_type_eia860_hydro_strings](#) (in module [pudl.constants](#)), 114
[fuel_type_eia860_nuclear_strings](#) (in module [pudl.constants](#)), 114
[fuel_type_eia860_oil_strings](#) (in module [pudl.constants](#)), 115
[fuel_type_eia860_other_strings](#) (in module [pudl.constants](#)), 115
[fuel_type_eia860_simple_map](#) (in module [pudl.constants](#)), 115
[fuel_type_eia860_solar_strings](#) (in module [pudl.constants](#)), 115
[fuel_type_eia860_waste_strings](#) (in module [pudl.constants](#)), 115
[fuel_type_eia860_wind_strings](#) (in module [pudl.constants](#)), 115
[fuel_type_eia923](#) (in module [pudl.constants](#)), 115
[fuel_type_eia923_boiler_fuel_coal_strings](#) (in module [pudl.constants](#)), 115
[fuel_type_eia923_boiler_fuel_gas_strings](#) (in module [pudl.constants](#)), 115
[fuel_type_eia923_boiler_fuel_oil_strings](#) (in module [pudl.constants](#)), 115
[fuel_type_eia923_boiler_fuel_other_strings](#) (in module [pudl.constants](#)), 115
[fuel_type_eia923_boiler_fuel_simple_map](#) (in module [pudl.constants](#)), 115
[fuel_type_eia923_boiler_fuel_waste_strings](#) (in module [pudl.constants](#)), 115
[fuel_type_eia923_gen_fuel_coal_strings](#) (in module [pudl.constants](#)), 116
[fuel_type_eia923_gen_fuel_gas_strings](#) (in module [pudl.constants](#)), 116
[fuel_type_eia923_gen_fuel_hydro_strings](#) (in module [pudl.constants](#)), 116
[fuel_type_eia923_gen_fuel_nuclear_strings](#) (in module [pudl.constants](#)), 116
[fuel_type_eia923_gen_fuel_oil_strings](#) (in module [pudl.constants](#)), 116
[fuel_type_eia923_gen_fuel_other_strings](#) (in module [pudl.constants](#)), 116
[fuel_type_eia923_gen_fuel_simple_map](#) (in module [pudl.constants](#)), 116
[fuel_type_eia923_gen_fuel_solar_strings](#) (in module [pudl.constants](#)), 116
[fuel_type_eia923_gen_fuel_waste_strings](#) (in module [pudl.constants](#)), 116
[fuel_type_eia923_gen_fuel_wind_strings](#) (in module [pudl.constants](#)), 116
[fuel_units_eia923](#) (in module [pudl.constants](#)), 116

G

[gen_eia923\(\)](#) ([pudl.output.pudltabl.PudlTabl](#) method), 84
[generate_data_packages\(\)](#) (in module [pudl.etl](#)), 120
[generate_metadata\(\)](#) (in module [pudl.load.metadata](#)), 72
[generation\(\)](#) (in module [pudl.transform.eia923](#)), 89
[generation_eia923\(\)](#) (in module [pudl.output.eia923](#)), 79
[generation_fuel\(\)](#) (in module [pudl.transform.eia923](#)), 90
[generation_fuel_eia923\(\)](#) (in module [pudl.output.eia923](#)), 79
[generation_fuel_map_eia923](#) (in module [pudl.constants](#)), 116
[generator_assn_map_eia860](#) (in module [pudl.constants](#)), 116
[generator_map_eia923](#) (in module [pudl.constants](#)), 117
[generator_proposed_assn_map_eia860](#) (in module [pudl.constants](#)), 117
[generator_retired_assn_map_eia860](#) (in module [pudl.constants](#)), 117
[generators\(\)](#) (in module [pudl.transform.eia860](#)), 87
[generators_eia860\(\)](#) (in module [pudl.output.eia860](#)), 76
[gens_eia860\(\)](#) ([pudl.output.pudltabl.PudlTabl](#) method), 84
[get_all_sources\(\)](#) (in module [pudl.convert.flatten_datapkg](#)), 53
[get_autoincrement_columns\(\)](#) (in module [pudl.load.metadata](#)), 72
[get_dbc_map\(\)](#) (in module [pudl.extract.ferc1](#)), 64
[get_dbf_path\(\)](#) (in module [pudl.extract.ferc1](#)), 65
[get_defaults\(\)](#) (in module [pudl.workspace.setup](#)), 104
[get_dependent_tables_from_list_pkg\(\)](#) (in module [pudl.load.metadata](#)), 72
[get_dependent_tables_pkg\(\)](#) (in module [pudl.load.metadata](#)), 73
[get_eia860_column_map\(\)](#) (in module [pudl.extract.eia860](#)), 54

`get_eia860_file()` (in module `pudl.extract.eia860`), 54
`get_eia860_page()` (in module `pudl.extract.eia860`), 54
`get_eia860_xlsx()` (in module `pudl.extract.eia860`), 55
`get_eia923_column_map()` (in module `pudl.extract.eia923`), 56
`get_eia923_file()` (in module `pudl.extract.eia923`), 56
`get_eia923_page()` (in module `pudl.extract.eia923`), 56
`get_eia923_xlsx()` (in module `pudl.extract.eia923`), 57
`get_epaipm_file()` (in module `pudl.extract.epaipm`), 58
`get_epaipm_name()` (in module `pudl.extract.epaipm`), 58
`get_ferc1_meta()` (in module `pudl.extract.ferc1`), 65
`get_flattened_etl_parameters()` (in module `pudl.etl`), 120
`get_foreign_key_relash_from_pkg()` (in module `pudl.load.metadata`), 73
`get_raw_df()` (in module `pudl.extract.ferc1`), 65
`get_repartitioned_tables()` (in module `pudl.load.metadata`), 73
`get_same_source_meta()` (in module `pudl.convert.flatten_datapkg`), 53
`get_source_metadata()` (in module `pudl.load.metadata`), 73
`get_strings()` (in module `pudl.extract.ferc1`), 65
`get_table_meta()` (in module `pudl.output.pudltabl`), 86
`get_tabular_data_resource()` (in module `pudl.load.metadata`), 73
`get_unpartitioned_tables()` (in module `pudl.load.metadata`), 74
`gf_eia923()` (`pudl.output.pudltabl.PudlTabl` method), 84
`gf_eia923_agg` (in module `pudl.validate`), 130
`gf_eia923_coal_heat_content` (in module `pudl.validate`), 130
`gf_eia923_gas_heat_content` (in module `pudl.validate`), 130
`gf_eia923_oil_heat_content` (in module `pudl.validate`), 130
`glue()` (in module `pudl.glue.ferc1_eia`), 68
`glue_pudl_tables` (in module `pudl.constants`), 117
H
`harmonize_eia_epa_orispl()` (in module `pudl.transform.epacems`), 91
`hash_csv()` (in module `pudl.load.metadata`), 74
`heat_rate_by_gen()` (in module `pudl.analysis.mcoe`), 48
`heat_rate_by_unit()` (in module `pudl.analysis.mcoe`), 48
`historical_distribution()` (in module `pudl.validate`), 130
`historical_histogram()` (in module `pudl.validate`), 131
`hr_by_gen()` (`pudl.output.pudltabl.PudlTabl` method), 84
`hr_by_unit()` (`pudl.output.pudltabl.PudlTabl` method), 84
I
`init()` (in module `pudl.workspace.setup`), 105
`initialize_parser()` (in module `pudl.workspace.setup_cli`), 106
`is_annual()` (in module `pudl.helpers`), 124
L
`licenses` (in module `pudl.constants`), 117
`load_curves()` (in module `pudl.transform.epaipm`), 92
M
`main()` (in module `pudl.cli`), 106
`main()` (in module `pudl.convert.datapkg_to_sqlite`), 49
`main()` (in module `pudl.convert.epacems_to_parquet`), 51
`main()` (in module `pudl.convert.ferc1_to_sqlite`), 51
`main()` (in module `pudl.workspace.datastore_cli`), 103
`main()` (in module `pudl.workspace.setup_cli`), 106
`make_ferc_clf()` (in module `pudl.transform.ferc1`), 96
`mcoe()` (in module `pudl.analysis.mcoe`), 48
`mcoe()` (`pudl.output.pudltabl.PudlTabl` method), 84
`mcoe_coal_capacity_factor` (in module `pudl.validate`), 131
`mcoe_coal_heat_rate` (in module `pudl.validate`), 131
`mcoe_fuel_cost_per_mmbtu` (in module `pudl.validate`), 131
`mcoe_fuel_cost_per_mwh` (in module `pudl.validate`), 131
`mcoe_gas_capacity_factor` (in module `pudl.validate`), 131
`mcoe_gas_heat_rate` (in module `pudl.validate`), 131
`merge_dicts()` (in module `pudl.helpers`), 125
`merge_on_date_year()` (in module `pudl.helpers`), 125
`missing_respondents_ferc1` (in module `pudl.constants`), 117
`month_dict_eia923` (in module `pudl.constants`), 117

`month_year_to_date()` (in module `pudl.helpers`), 125

N

`need_fix_inting` (in module `pudl.constants`), 117

`nerc_region` (in module `pudl.constants`), 117

O

`organize()` (in module `pudl.workspace.datastore`), 101

`organize_cols()` (in module `pudl.helpers`), 126

`output_formats` (in module `pudl.constants`), 117

`own_eia860()` (`pudl.output.pudltabl.PudlTabl` method), 85

`ownership()` (in module `pudl.transform.eia860`), 87

`ownership_assn_map_eia860` (in module `pudl.constants`), 117

`ownership_eia860()` (in module `pudl.output.eia860`), 76

P

`package_files_from_table()` (in module `pudl.load.metadata`), 74

`parallel_update()` (in module `pudl.workspace.datastore`), 101

`parse_command_line()` (in module `pudl.cli`), 106

`parse_command_line()` (in module `pudl.convert.datapkg_to_sqlite`), 49

`parse_command_line()` (in module `pudl.convert.epacems_to_parquet`), 51

`parse_command_line()` (in module `pudl.convert.ferc1_to_sqlite`), 51

`parse_command_line()` (in module `pudl.workspace.datastore_cli`), 104

`parseN()` (`pudl.extract.ferc1.FERC1FieldParser` method), 59

`path()` (in module `pudl.workspace.datastore`), 101

`paths_for_year()` (in module `pudl.workspace.datastore`), 102

`pkg_to_sqlite_db()` (in module `pudl.convert.datapkg_to_sqlite`), 49

`plant_assn_map_eia860` (in module `pudl.constants`), 117

`plant_frame_map_eia923` (in module `pudl.constants`), 117

`plant_in_service()` (in module `pudl.extract.ferc1`), 66

`plant_in_service()` (in module `pudl.transform.ferc1`), 97

`plant_region_map()` (in module `pudl.transform.epaipm`), 92

`plants()` (in module `pudl.transform.eia860`), 88

`plants()` (in module `pudl.transform.eia923`), 90

`plants_eia860()` (in module `pudl.output.eia860`), 76

`plants_eia860()` (`pudl.output.pudltabl.PudlTabl` method), 85

`plants_hydro()` (in module `pudl.extract.ferc1`), 66

`plants_hydro()` (in module `pudl.transform.ferc1`), 97

`plants_pumped_storage()` (in module `pudl.extract.ferc1`), 66

`plants_pumped_storage()` (in module `pudl.transform.ferc1`), 97

`plants_small()` (in module `pudl.extract.ferc1`), 66

`plants_small()` (in module `pudl.transform.ferc1`), 98

`plants_steam()` (in module `pudl.extract.ferc1`), 67

`plants_steam()` (in module `pudl.transform.ferc1`), 98

`plants_steam_ferc1()` (in module `pudl.output.ferc1`), 81

`plants_steam_ferc1()` (`pudl.output.pudltabl.PudlTabl` method), 85

`plants_steam_validate_ids()` (in module `pudl.transform.ferc1`), 98

`plants_utils_eia860()` (in module `pudl.output.eia860`), 77

`plants_utils_ferc1()` (in module `pudl.output.ferc1`), 81

`plot_vs_agg()` (in module `pudl.validate`), 131

`plot_vs_bounds()` (in module `pudl.validate`), 131

`plot_vs_self()` (in module `pudl.validate`), 131

`predict()` (`pudl.transform.ferc1.FERCPlantClassifier` method), 94

`prep_pkg_bundle_directory()` (in module `pudl.load.metadata`), 74

`prime_movers` (in module `pudl.constants`), 118

`prime_movers_eia923` (in module `pudl.constants`), 118

`pu_eia860()` (`pudl.output.pudltabl.PudlTabl` method), 85

`pu_ferc1()` (`pudl.output.pudltabl.PudlTabl` method), 85

`pudl` (module), 132

`pudl.analysis` (module), 49

`pudl.analysis.mcoe` (module), 48

`pudl.cli` (module), 106

`pudl.constants` (module), 107

`pudl.convert` (module), 53

`pudl.convert.datapkg_to_sqlite` (module), 49

`pudl.convert.epacems_to_parquet` (module), 50

`pudl.convert.ferc1_to_sqlite` (module), 51

`pudl.convert.flatten_datapkg` (module), 51

`pudl.etl` (module), 119

`pudl.extract` (module), 67

[pudl.extract.eia860 \(module\), 53](#)
[pudl.extract.eia923 \(module\), 55](#)
[pudl.extract.epacems \(module\), 57](#)
[pudl.extract.epaipm \(module\), 58](#)
[pudl.extract.ferc1 \(module\), 59](#)
[pudl.glue \(module\), 69](#)
[pudl.glue.ferc1_eia \(module\), 68](#)
[pudl.helpers \(module\), 121](#)
[pudl.load \(module\), 75](#)
[pudl.load.csv \(module\), 69](#)
[pudl.load.metadata \(module\), 71](#)
[pudl.output \(module\), 86](#)
[pudl.output.eia860 \(module\), 75](#)
[pudl.output.eia923 \(module\), 77](#)
[pudl.output.export \(module\), 80](#)
[pudl.output.ferc1 \(module\), 80](#)
[pudl.output.glue \(module\), 82](#)
[pudl.output.pudltabl \(module\), 82](#)
[pudl.transform \(module\), 99](#)
[pudl.transform.eia \(module\), 86](#)
[pudl.transform.eia860 \(module\), 87](#)
[pudl.transform.eia923 \(module\), 89](#)
[pudl.transform.epacems \(module\), 91](#)
[pudl.transform.epaipm \(module\), 92](#)
[pudl.transform.ferc1 \(module\), 93](#)
[pudl.validate \(module\), 129](#)
[pudl.workspace \(module\), 106](#)
[pudl.workspace.datastore \(module\), 99](#)
[pudl.workspace.datastore_cli \(module\), 103](#)
[pudl.workspace.setup \(module\), 104](#)
[pudl.workspace.setup_cli \(module\), 105](#)
[pudl_tables \(in module pudl.constants\), 118](#)
[PudlTabl \(class in pudl.output.pudltabl\), 82](#)
[pull_resource_from_megadata\(\) \(in module pudl.load.metadata\), 74](#)
[purchased_power\(\) \(in module pudl.extract.ferc1\), 67](#)
[purchased_power\(\) \(in module pudl.transform.ferc1\), 98](#)
 Python Enhancement Proposals
 [PEP 257, 28](#)
 [PEP 517, 34](#)
 [PEP 518, 34](#)
 [PEP 8, 28, 42](#)

R

[read_cems_csv\(\) \(in module pudl.extract.epacems\), 58](#)
[read_excel_epaipm_dict \(in module pudl.constants\), 118](#)
[rto_iso \(in module pudl.constants\), 118](#)

S

[score\(\) \(pudl.transform.ferc1.FERCPlantClassifier](#)

[method\), 94](#)

[sector_eia \(in module pudl.constants\), 118](#)
[set_defaults\(\) \(in module pudl.workspace.setup\), 105](#)
[show_dupes\(\) \(in module pudl.extract.ferc1\), 67](#)
[simplify_columns\(\) \(in module pudl.helpers\), 126](#)
[skiprows_eia860 \(in module pudl.constants\), 118](#)
[skiprows_eia923 \(in module pudl.constants\), 118](#)
[source_url\(\) \(in module pudl.workspace.datastore\), 102](#)
[spill\(\) \(pudl.load.csv.BulkCopy method\), 70](#)
[state_tz_approx \(in module pudl.constants\), 118](#)
[stocks_map_eia923 \(in module pudl.constants\), 118](#)
[strip_lower\(\) \(in module pudl.helpers\), 126](#)
[sum_na\(\) \(in module pudl.helpers\), 127](#)

T

[tab_map_eia860 \(in module pudl.constants\), 118](#)
[tab_map_eia923 \(in module pudl.constants\), 118](#)
[table_map_ferc1_pudl \(in module pudl.constants\), 118](#)
[test_file_consistency\(\) \(in module pudl.load.metadata\), 75](#)
[transform\(\) \(in module pudl.transform.eia\), 86](#)
[transform\(\) \(in module pudl.transform.eia860\), 88](#)
[transform\(\) \(in module pudl.transform.eia923\), 90](#)
[transform\(\) \(in module pudl.transform.epacems\), 91](#)
[transform\(\) \(in module pudl.transform.epaipm\), 92](#)
[transform\(\) \(in module pudl.transform.ferc1\), 99](#)
[transform\(\) \(pudl.transform.ferc1.FERCPlantClassifier method\), 94](#)
[transmission_joint\(\) \(in module pudl.transform.epaipm\), 92](#)
[transmission_single\(\) \(in module pudl.transform.epaipm\), 93](#)
[transport_modes_eia923 \(in module pudl.constants\), 119](#)
[travis_ci_eia860_years \(in module pudl.constants\), 119](#)
[travis_ci_eia923_years \(in module pudl.constants\), 119](#)
[travis_ci_epacems_states \(in module pudl.constants\), 119](#)
[travis_ci_epacems_years \(in module pudl.constants\), 119](#)
[travis_ci_ferc1_years \(in module pudl.constants\), 119](#)

U

[update\(\) \(in module pudl.workspace.datastore\), 103](#)
[us_states \(in module pudl.constants\), 119](#)
[utilities\(\) \(in module pudl.transform.eia860\), 88](#)
[utilities_eia860\(\) \(in module pudl.output.eia860\), 77](#)

`utility_assn_map_eia860` (in module `pudl.constants`), 119
`utils_eia860()` (`pudl.output.pudltabl.PudlTabl` method), 85

V

`validate_params()` (in module `pudl.etl`), 121
`validate_save_pkg()` (in module `pudl.load.metadata`), 75
`verify_input_files()` (in module `pudl.helpers`), 128
`vs_bounds()` (in module `pudl.validate`), 131
`vs_historical()` (in module `pudl.validate`), 131
`vs_self()` (in module `pudl.validate`), 131

W

`weighted_quantile()` (in module `pudl.validate`), 132
`working_years` (in module `pudl.constants`), 119

X

`xlsx_maps_pkg` (in module `pudl.constants`), 119

Y

`year_from_operating_datetime()` (in module `pudl.convert.epacems_to_parquet`), 51