PUDL

Release 0.3.3.dev992+gd8d9103

Catalyst Cooperative

Apr 30, 2021

CONTENTS

:

CHAPTER

ONE

WHAT IS PUDL?

The PUDL Project is an open source data processing pipeline that makes US energy data easier to access and use programmatically.

Hundreds of gigabytes of valuable data are published by US government agencies, but it's often difficult to work with. PUDL takes the original spreadsheets, CSV files, and databases and turns them into a unified resource. This allows users to spend more time on novel analysis and less time on data preparation.

CHAPTER

TWO

WHAT DATA IS AVAILABLE?

PUDL currently integrates data from:

- EIA Form 860 (2004-2019)
- EIA Form 860m (2020-2021)
- EIA Form 861 (2001-2019)
- EIA Form 923 (2009-2019)
- EPA Continuous Emissions Monitoring System (CEMS) (1995-2020)
- FERC Form 1 (1994-2019)
- FERC Form 714 (2006-2019)
- US Census Demographic Profile 1 Geodatabase (2010)

Thanks to support from the Alfred P. Sloan Foundation Energy & Environment Program, from 2021 to 2023 we will be integrating the following data as well:

- EIA Form 176 (The Annual Report of Natural Gas Supply and Disposition)
- FERC Electric Quarterly Reports (EQR)
- FERC Form 2 (Annual Report of Major Natural Gas Companies)
- PHMSA Natural Gas Annual Report
- Machine Readable Specifications of State Clean Energy Standards

CHAPTER THREE

WHO IS PUDL FOR?

The project is focused on serving researchers, activists, journalists, policy makers, and small businesses that might not otherwise be able to afford access to this data from commercial sources and who may not have the time or expertise to do all the data processing themselves from scratch.

We want to make this data accessible and easy to work with for as wide an audience as possible: anyone from a grassroots youth climate organizers working with Google sheets to university researchers with access to scalable cloud computing resources and everyone in between!

CHAPTER

FOUR

HOW DO I ACCESS THE DATA?

There are four main ways to access PUDL outputs. For more details you'll want to check out the complete documentation, but here's a quick overview:

4.1 Datasette

We publish a lot of the data on https://data.catalyst.coop using a tool called Datasette which lets us wrap our databases in a relatively friendly web interface. You can browse and query the data, make simple charts and maps, and download portions of the data as CSV files or JSON so you can work with it locally. For a quick introduction to what you can do with the Datasette interface, check out this 17 minute video.

This access mode is good for casual data explorers or anyone who just wants to grab a small subset of the data. It also lets you share links to a particular subset of the data and provides a REST API for querying the data from other applications.

4.2 Docker + Jupyter

Want access to all the published data in bulk? If you're familiar with Python and Jupyter Notebooks and are willing to install Docker you can:

- Download a PUDL data release from CERN's Zenodo archiving service.
- Install Docker
- Run the archived image using `docker-compose up`
- Access the data via the resulting Jupyter Notebook server running on your machine.

If you'd rather work with the PUDL SQLite Databases and Apache Parquet files directly, they are accessible within the same Zenodo archive.

The PUDL Examples repository has more detailed instructions on how to work with the Zenodo data archive and Docker image.

4.3 JupyterHub

Do you want to use Python and Jupyter Notebooks to access the data but aren't comfortable setting up Docker? We are working with 2i2c to host a JupyterHub that has the same software and data as the Docker container and Zenodo archive mentioned above, but running in the cloud.

- Request an account
- Log in to the JupyterHub

Note: you'll only have 4-6GB of RAM and 1 CPU to work with on the JupyterHub, so if you need more computing power, you may need to set PUDL up on your own computer. Eventually we hope to offer scalable computing resources on the JupyterHub as well.

4.4 The PUDL Development Environment

If you're more familiar with the Python data science stack and are comfortable working with git, `conda` environments, and the Unix command line, then you can set up the whole PUDL Development Environment on your own computer. This will allow you to run the full data processing pipeline yourself, tweak the underlying source code, and (we hope!) make contributions back to the project.

This is by far the most involved way to access the data and isn't recommended for most users. You should check out the Development section of the main PUDL documentation for more details.

CHAPTER FIVE

CONTRIBUTING TO PUDL

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

- First, 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 allow us to integrate the resulting code into PUDL.
- For more information check out the Contributing section of the PUDL Documentation

CHAPTER

SIX

LICENSING

In general, our code, data, and other work are permissively licensed for use by anybody, for any purpose, so long as you give us credit for the work we've done.

- The PUDL software is released under the MIT License.
- The PUDL data and documentation are published under the Creative Commons Attribution License v4.0 (CC-BY-4.0).

CHAPTER

SEVEN

CONTACT US

- For user support, bug reports and anything else that could be useful or interesting to other users, please make a GitHub issue.
- For private communication about the project, you can email the maintainers: pudl@catalyst.coop
- If you'd like to get occasional updates about the project sign up for our email list.
- Follow us on Twitter: @CatalystCoop
- More info on our website: https://catalyst.coop

CHAPTER

EIGHT

ABOUT CATALYST COOPERATIVE

Catalyst Cooperative is a small group of data wranglers and policy wonks 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 (Hire us!). Our focus is primarily on mitigating climate change and improving electric utility regulation in the United States.

8.1 Introduction

PUDL is a data processing pipeline that cleans, integrates, and standardizes some of the most widely used public energy datasets in the US. The data serve researchers, activists, journalists, and policy makers that might not have the means to purchase processed data from existing commercial providers, the technical expertise to access it in its raw form, or the time to clean and prepare the data for bulk analysis.

8.1.1 Available Data

Currently, PUDL has integrated data from:

- *EIA Form 860* (including EIA 860m)
- *EIA Form 861* (preliminary)
- *EIA Form 923*
- *FERC Form 1*
- *FERC Form 714* (preliminary)
- *EPA CEMS Hourly*

In addition, we distribute an SQLite databases containing all available years of the raw FERC Form 1 data and an SQLite version of the US Census DP1 geodatabase

If you want to get started using PUDL data, visit our *Data Access* page. Read on to learn about the components of the data processing pipeline.

8.1.2 Raw Data Archives

Because the original data PUDL depends on frequently changes, and old versions don't remain available, we periodically create archives of the raw inputs on Zenodo. They are issued DOIs and made available via Zenodo's REST API. Each data source can have several different versions, each with its own unique DOI. Each release of the PUDL Python package has a set of DOIs embedded in it, indicating which version of the raw inputs it is meant to process. This helps ensure that our outputs are replicable.

These raw inputs are organized into Frictionless Data Packages with some extra metadata indicating how they are partitioned (by year, state, etc.). The format of the underlying data varies from source to source, and in some cases from year to year, and includes CSVs, Excel spreadsheets, and Visual FoxPro database (DBF) files.

The PUDL software will download a copy of the appropriate raw inputs automatically as needed, and organizes them in a local *datastore*.

See also:

The software that creates and archives the raw inputs can be found in our PUDL Scrapers and PUDL Zenodo Storage repositories on GitHub.

8.1.3 The ETL Process

The core of PUDL's work takes place in the ETL (Extract, Transform, and Load) process.

Extract

The Extract step reads the raw data from its original heterogeneous formats into a collection of `pandas.DataFrame` with uniform column names across all years, so that it can be easily processed in bulk. In the case of data distributed as binary database files like DBF such as the FERC Form 1, it may be converted into a unified SQLite database before individual dataframes are created.

See also:

Module documentation within the `pudl.extract` subpackage.

Transform

The Transform step is generally broken down into two phases. The first focuses on cleaning and organizing data within individual tables, and the focuses on the integration and deduplication of data between tables. These tasks can be tedious data wrangling toil that impose a huge amount of overhead on anyone trying to do analysis based on the publicly available data. PUDL implements common data cleaning operations in the hopes that we can all work on more interesting problems most of the time. These operations include:

- Standardization of units (e.g. dollars not thousands of dollars)
- Standardization of N/A values
- Standardization of freeform names and IDs
- Use of controlled vocabularies for categorical values like fuel type
- Use of more readable codes and column names
- Imposition of well defined, rich data types for each column
- Converting local timestamps to UTC
- Reshaping of data into well normalized tables which minimize data duplication

- Inferring Plant IDs which link records across many years of FERC Form 1 data
- Inferring linkages between FERC and EIA Plants and Utilities.
- Inferring more complete associations between EIA boilers and generators

See also:

The module and per-table transform functions in the *pudl.transform* sub-package have more details on the specific transformations applied to each table.

Many of the original datasets contain large amounts of duplicated data. For instance, the EIA reports the name of each power plant in every table that refers to otherwise unique plant-related data. Similarly, many attributes like plant latitude and longitude are reported separately every year. Often these reported values are not self-consistent. There may be several different spellings of a plant's name, or an incorrectly reported latitude in one year.

The transform step attempts to eliminate this kind of inconsistent duplicate information when normalizing the tables, choosing only the most consistently reported value for inclusion in the final database. If a value which should be static is not consistently reported, it may also be set to N/A.

See also:

- Tidy Data by Hadley Wickham, Journal of Statistical Software (2014).
- A Simple Guide to the Five Normal Forms in Relational Database Theory by William Kent, Communications of the ACM (1983).

Load

At the end of the Transform step, we have collections of DataFrames which correspond to database tables. These written out to ("loaded" into) platform indepenent tabular data packages where the data is stored as CSV files, and the metadata is stored as JSON. These sttatic, text-based output formats are archive-friendly, and can be used to populate a database, or read with Python, R, and many other tools.

Note: Starting with v0.5.0 of PUDL, we will begin generating SQLite database and Apache Parquet file outputs directly, and using those formats to distribute the processed data.

See also:

Module documentation within the *pudl.load* sub-package.

8.1.4 Database & Output Tables

Tabular Data Packages are archive friendly and platform independent, but given the size and complexity of the data within PUDL, this format isn't ideal for day to day interactive use. In practice, we take the clean, processed data in the data packages and use it to populate a local SQLite database. To handle the ~1 billion row EPA CEMS hourly time series we convert the data package into Apache Parquet dataset which is partitioned by state and year. For more details on these conversions to SQLite and Parquet formats, see *Data Packages*.

Denormalized Outputs

Working with the PUDL data interactively, you'll often want to combine information from more than one table to make the data more readable and readily interpretable. For example the name that EIA uses to refer to a power plant is only stored in the *plants_entity_eia* table in association with the plant's unique numeric ID. If you are working with data from the *fuel_receipts_costs_eia923* table, which records monthly per-plant fuel deliveries, you may want to have the name of the plant alongside the fuel delivery information since it's more recognizable than the plant ID.

Rather than requiring everyone to write their own SQL `SELECT` and `JOIN` statements or do a bunch of `pandas.merge()` operations to bring together data, PUDL provides a variety of predefined queries as methods of the *`pudl.output.pudltabl.PudlTabl`* class, which do common joins and return dataframes that are convenient for interactive use. This avoids duplicating data in the database (which often leads to data integrity issues), but still provides convenient user access.

Note: In the future we intend to replace the simple denormalized output tables with database views which are integrated into the distributed SQLite database directly. This will provide the same convenience without requiring use of the Python software layer.

Analysis Outputs

There are several analytical routines built into the *`pudl.output.pudltabl.PudlTabl`* output objects for calculating derived values like the heat rate by generation unit (*`hr_by_unit`*), or the capacity factor by generator (*`capacity_factor`*). We intend to integrate more analytical output into the library over time.

See also:

- The PUDL Examples GitHub repo to see how to access the PUDL Database directly, use the output functions, or work with the EPA CEMS data using Dask.
- How to Learn Dask in 2021 is a great collection of self-guided resources if you are already familiar with Python, Pandas, and NumPy.

8.1.5 Data Validation

We have a growing collection of data validation test cases which we run before publishing a data release to try and avoid publishing data wth known issues. Most of these validations are described in the *`pudl.validate`* module. They check things like:

- The heat content of various fuel types are within expected bounds.
- Coal ash, moisture, mercury, sulfur etc. content are within expected bounds
- Generator heat rates and capacity factors are realistic for the type of prime mover being reported.

Some data validations are currently only specified within our test suite, including:

- The expected number of records within each table
- The fact that there are no entirely N/A columns

A variety of database integrity checks are also run either during the ETL process or when the data is loaded into SQLite.

See our *Testing PUDL* documentation for more information.

8.2 Data Access

We publish the *PUDL pipeline* outputs in several ways to serve different users and use cases. We're always trying to increase accessibility of the PUDL data, so if you have suggestions or questions please open a GitHub issue or email us at pudl@catalyst.coop.

8.2.1 How Should You Access PUDL Data?

We provide four primary ways of interacting with PUDL data. Here's how to find out which one is right for you and your use case.

Access Method	Types of User	Use Cases
Datasette	Curious Explorer, Spreadsheet Analyst, Web Developer	Explore the PUDL database interactively in a web browser. Select data to download as CSVs for local analysis in spreadsheets. Create sharable links to a particular selection of data. Access PUDL data via a REST API.
Zenodo Archives	Researcher, Database User, Notebook Analyst	Use a stable, citable, fully processed version of the PUDL on your own computer. Use PUDL in Jupyer Notebooks running in a stable, archived Docker container. Access the SQLite DB and Parquet files directly using any toolset.
JupyterHub	New Python User, Notebook Analyst	Work through the PUDL example notebooks ithout any downloads or setup. Perform your own notebook-based analyses using PUDL data and limited computational resources.
Development Environment	Python Developer, Data Wrangler	Run the PUDL data processing pipeline on your own computer. Edit the PUDL source code and run the software tests and data validations. Integrate a new data source or newly released data from one of existing sources.
Data Packages	Deprecated	For working with our published data prior to v0.4.0

8.2.2 Datasette

We provide web-based access to the PUDL data via a Datasette deployment at https://data.catalyst.coop.

Datasette is an open source tool that wraps SQLite databases in an interactive front-end. It allows users to browse database tables, select portions of them using dropdown menus, build their own SQL queries, and download data to CSVs. It also creates a REST API allowing the data in the database to be queried programmatically. All the query parameters are stored in the URL, so you can also share links to the data you've selected.

Note that only data which has been fully integrated into the SQLite databases are available here. Currently this includes the core PUDL database and our concatenation of all historical FERC Form 1 databases.

8.2.3 Zenodo Archives

We use Zenodo to archive our fully processed data as a SQLite databasees and Parquet files. We also archive a Docker image that contains the software environment required to use PUDL within Jupyter Notebooks. You can find all our archived data products in the Catalyst Cooperative Community on Zenodo.

- The current (beta) version of the archived data and Docker container can be downloaded from This Zenodo archive
- Detailed instructions on how to access the archived PUDL data using a Docker container can be found in our PUDL Examples repository.
- The SQLite databases and Parquet files containing the PUDL data, the complete FERC 1 database, and EPA CEMS hourly data are contained in that same archive, if you want to access them directly without using PUDL.

Note: If you're already familiar with Docker, you can also pull the image we use to run Jupyter directly:

```
$ docker pull catalystcoop/pudl-jupyter:latest
```

8.2.4 JupyterHub

We've set up a JupyterHub in collaboration with 2i2c.org which provides access to all of the processed PUDL data and the software environment required to work with it. You don't have to download or install anything to use it, but we do need to create an account for you.

- Request an account by submitting this form.
- Once we've created an account for you follow this link to log in and open up the first example notebook on the JupyterHub.
- You can create your own notebooks and upload, save, and download modest amounts of data on the hub.

We can only offer a small amount of memory (4-6GB) and processing power (1 CPU) per user on the JupyterHub for free. If you need to work with lots of data or do computationally intensive analysis, you may want to look into using the *Zenodo Archives* option on your own computer. The JupyterHub uses exactly the same data and software environment as the Zenodo Archives. Eventually we also want to offer paid access to the JupyterHub with plenty of computing power.

8.2.5 Development Environment

If you want to run the PUDL data processing pipeline yourself from scratch, run the software tests, or make changes to the source code, you'll need to set up our development environment. This is a bit involved, so it has its *own separate documentation*.

Most users shouldn't need to do this, and will probably find working with the pre-processed data via one of the other access modes easier. But if you want to *contribute to the project* please give it a shot!

8.2.6 Data Packages

Note: Prior to v0.4.0 of PUDL we only published processed data as tabular data packages. As of v0.4.0 we are will distribute the SQLite databases and Apache Parquet files alongside a set of data packages. As of PUDL v0.5.0 we will be generating SQLite and Apache Parquet outputs directly, and will no longer be archiving tabular data packages as the format of record, and the format conversions described below will no longer be necessary.

Archived Data Packages

We periodically publish data packages containing the full outputs from the PUDL ETL pipeline on Zenodo, and open data archiving service provided by CERN. The most recent release can always be found through this concept DOI: 10.5281/zenodo.3653158. Each individual version of the data releases will be assigned its own unique DOI.

All of our archived products can be found in the Catalyst Cooperative Community on Zenodo. These archives and the DOIs associated with them should be permanently accessible, and are suitable for use as references in academic and other publications.

Once you've downloaded or generated your own tabular data packages you will probably want to convert them into a more analysis oriented file format. We typically use SQLite for the core FERC and EIA data, and Apache Parquet files for the very long tables like EPA CEMS.

Converting to SQLite

If you want to access the data via SQL, we have provided a script that loads several data packages into a local sqlite3 database. Note that these data packages **must** have all been generated by the **same** ETL run, or they will be considered incompatible by the script. For example, to load three data packages generated by our example ETL configuration into your local SQLite DB, you could run the following command from within your PUDL workspace:

```
$ datapkg_to_sqlite \
    datapkg/pudl-example/ferc1-example/datapackage.json \
    datapkg/pudl-example/eia-example/datapackage.json \
```

Run `datapkg_to_sqlite --help` for more details.

Converting to Apache Parquet

The *EPA CEMS Hourly* data approaches 100 GB in size uncompressed, which is too large to work with directly in memory on most systems, and take a very long time to load into SQLite. Instead, we recommend converting the Hourly Emissions table into an Apache Parquet dataset which is stored on disk locally, and either reading in only parts of it using pandas, or using Dask dataframes, to serialize or distribute your analysis tasks. Dask 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.

If you have generated an EPA CEMS data package, you can use the `epacems_to_parquet` script to convert the hourly emissions table like this:

```
$ epacems_to_parquet datapkg/pudl-example/epacems-eia-example/datapackage.json
```

The script will automatically generate a Parquet Dataset which is partitioned by year and state in the `parquet/epacems` directory within your workspace. Run `epacems_to_parquet --help` for more details.

8.3 Data Sources

8.3.1 EIA Form 860

Source URL	https://www.eia.gov/electricity/data/eia860/
Source Description	The status of existing electric generating plants and associated equipment in the United States, and those scheduled for initial commercial operation within 10 years of the filing.
Respondents	Utilities
Source Format	Microsoft Excel (.xls/.xlsx)
Source Years	2001-2019
Size (Download)	413.4 MB
PUDL Code	`eia860`
Years Liberated	2004-2019
Records Liberated	~1 million
Issues	open EIA 860 issues

Background

The Form EIA-860 collects utility, owner, plant, and generator-level data from existing and planned entities with one or more megawatt of capacity. The form also contains information regarding environmental control equipment, and construction cost data from 2013-2018.

- `EIA-860 Instructions (PDF, to 2020-03-31)`
- `EIA-860 Instructions (PDF, to 2023-05-31)`

As of 2019, the EIA-860 Form is organized into the following schedules:

- **Schedule 1:** Identification
- **Schedule 2:** Power plant data
- **Schedule 3:** Generator information
- **Schedule 4:** Ownership of generators
- **Schedule 6:** Boiler information

(Schedule 5 contained generator construction cost information)

Who is required to fill out the form?

Respondents include all existing and proposed plants that have a total generator nameplate capacity (sum for generators at a single site) of 1 Megawatt (MW) or greater and are connected to the local or regional electric power grid. Annual responses are due between the beginning of January and the end of February.

Jointly owned plants must be reported only once by their operator or planned operator.

What does the original data look like?

Approximately a year after respondents submit their form, the EIA publishes the data in a series of spreadsheets that reflect the thematic contents of the form. These spreadsheets can change year-to-year as the questions in the form are updated and as EIA adopts new formatting standards for their outputs. They are accessible on the EIA website as downloadable ZIP files categorized by year. To gain greater insight into year-to-year nuances of the form, we recommend downloading multiple years of EIA-860 ZIP files and comparing both the Form and the Form Instructions files. See below for our description of notable irregularities in the data.

How much of the data is accessible through PUDL?

EIA-860 data stretches back to 2001, and PUDL currently covers all years starting from 2004. The prior years are published as DBF files and need a special process to read and extract. We intend to include these older years as soon as we can.

PUDL does not currently include the files pertaining to specific renewable energy resources or interconnection.

Notable Irregularities

In 2012 and 2013, the Form was updated to include specific information about renewable generators. These new data are not included in PUDL.

Prior to 2009, the Generators table was split into two spreadsheets, one for operating and one for proposed generation. In 2007 and before, there was an additional file for proposed changes to existing generation. The latter is excluded from PUDL while the former is combined into a single table during the transformation process.

EIA 860 includes a table in "Schedule 6: Boiler Information" which is an association table between boilers and generators. This association is important because in EIA 923 the net generation is reported by generators and the fuel consumption is reported by boilers - so a good boiler generator association is crucial for understanding heat rates. Unfortunately, the reported associations are incomplete. We have implemented a methodology fills in many of the missing links 2014 and later, and covers more than 95% net generation reported in the *generation_eia923* table. See this blog post and `pudl.transform.eia` for more information.

PUDL Data Tables

We've segmented the processed EIA-860 data into the following normalized data tables. Clicking on the links will show you the names and descriptions of the fields available in each table.

- *generators_eia860*
- *ownership_eia860*
- *boiler_generator_assn_eia860*
- *plants_eia860*
- *utilities_eia860*

We've also created the following entity tables modeled after EIA data collected from multiple tables

- *boilers_entity_eia*
- *generators_entity_eia*
- *plants_entity_eia*
- *utilities_entity_eia*

PUDL Data Transformations

The PUDL transformation process cleans the input data so that it is adjusted for uniformity, corrected for errors, and ready for bulk programmatic use.

To see the transformations applied to the data in each table, you can read the doc-strings for `pudl.transform.eia860` created for each tables' respective transform function.

8.3.2 EIA Form 923

Source URL	https://www.eia.gov/electricity/data/eia923/
Source Description	Generation, consumption, stocks, receipts
Respondents	Electric, CHP plants, and sometimes fuel transfer terminals with either 1MW+ or the ability to receive and deliver power to the grid.
Source Format	Microsoft Excel (.xls/.xlsx)
Source Years	2001-2019
Size (Download)	243.3 MB
PUDL Code	`eia923`
Years Liberated	2009-2019
Records Liberated	~3.6 million
Issues	Open EIA 923 issues

Background

Form EIA-923 is known as the **Power Plant Operations Report**. The data include electric power generation, energy source consumption, end of reporting period fossil fuel stocks, as well as the quality and cost of fossil fuel receipts at the power plant and prime mover level (with a subset of +10MW steam-electric plants reporting at the boiler and generator level. Information is available for non-utility plants starting in 1970 and utility plants beginning in 1999. The Form EIA-923 has evolved over the years, beginning as an environmental add-on in 2007 and ultimately eclipsing the information previously recorded in EIA-906, EIA-920, FERC 423, and EIA-423 by 2008.

- `EIA-923 Instructions (PDF, to 2020-03-31)`
- `EIA-923 Instructions (PDF, to 2023-05-31)`

As of 2019, the EIA-923 Form is organized into the following schedules:

- **Schedule 2:** fuel receipts and costs

- **Schedules 3A & 5A:** generator data including generation, fuel consumption and stocks
- **Schedule 4:** fossil fuel stocks
- **Schedules 6 & 7:** non-utility source and disposition of electricity
- **Schedules 8A-F:** environmental data

Who is required to fill out the form?

Respondents include all all electric and CHP plants, and in some cases fuel transfer terminals, that have a total generator nameplate capacity (sum for generators at a single site) of 1 Megawatt (MW) or greater and are connected to the local or regional electric power grid.

Selected plants may be permitted to report schedules 1-4B monthly and 6-8 annually so as to lighten their reporting burden. All other respondents must respond to the Form in its entirety once a year.

What does the original data look like?

Once the respondents have submitted their responses, the EIA creates a series of spreadsheets that reflect themes within the form. These spreadsheets have changed over the years as the form itself evolves. They are accessible on the EIA website as downloadable ZIP files categorized by year. The internal data are organized into excel spreadsheets. To gain greater insight into year-to-year nuances of the form, we recommend downloading multiple years of EIA-923 ZIP files and comparing both the Form and the Form Instructions files.

How much of the data is accessible through PUDL?

EIA-923 data stretches back to 1970, and PUDL currently covers all years starting from 2009. Due to a difference in reporting between the older and newer years, the older data will require more time to integrate. Monthly and year to date releases are not yet integrated.

In addition, We have not yet integrated tables reporting fuel stocks, data from Puerto Rico, or EIA-923 schedules 6, 7, and 8.

Notable Irregularities

File Naming Conventions

The naming conventions for the raw files are confusing and difficult to trace year to year. Subtle and not so subtle changes to the form and published spreadsheets make aggregating pre-2009 data difficult from a programmatic standpoint.

Protected Data

In accordance with the Freedom of Information Act and the Trade Secrets Act, certain information reported to EIA-923 may remain undisclosed to the public until three months after its collection date. The fields subject to this legislation include: total delivered cost of coal, natural gas, and petroleum received at non-utility power plants and the commodity cost information for all plants (Schedule 2).

Net generation & fuel consumed reported in two seperate tables

Net generation and fuel consumption are reported in two seperate tables in EIA-923: in the *generation_eia923* and *generation_fuel_eia923* tables. The *generation_fuel_eia923* table is more complete (the *generation_eia923* table includes only ~55% of the reported MWh), but the *generation_eia923* table is more granular (it is reported at the generator level).

Data Estimates

Plants that did not respond or reported unverified data were recorded as estimates rolled in with the state/fuel aggregates values reported under the plant id 99999.

PUDL Database Tables

We've segmented the processed EIA-923 data into the following normalized data tables. Clicking on the links will show you the names and descriptions of the fields available in each table.

EIA-923 Data Tables

These tables contain the bulk data reported in the EIA-923:

- *boiler_fuel_eia923*
- *coalmine_eia923*
- *fuel_receipts_costs_eia923*
- *generation_eia923*
- *generation_fuel_eia923*

EIA-923 Structural Tables

These tables define various codes and abbreviations more fully:

- *energy_source_eia923*
- *fuel_type_aer_eia923*
- *fuel_type_eia923*
- *prime_movers_eia923*
- *transport_modes_eia923*

PUDL Data Transformations

The PUDL transformation process cleans the input data so that it is adjusted for uniformity, corrected for errors, and ready for bulk programmatic use.

To see the transformations applied to the data in each table, you can read the function level documentation in `pudl.transform.eia923`.

8.3.3 EPA CEMS Hourly

Source URL	ftp://newftp.epa.gov/dmdnload/emissions/hourly/monthly
Source Description	Hourly CO2, SO2, NOx emissions and gross load
Respondents	Coal and high-sulfur fueled plants
Source Format	Comma Separated Value (.csv)
Source Years	1995-2019
Size (Download)	8.7 GB
PUDL Code	`epacems`
Years Liberated	1995-2019
Records Liberated	~1 billion
Issues	Open EPA CEMS issues

Background

As depicted by the EPA, Continuous Emissions Monitoring Systems (CEMS) are the “total equipment necessary for the determination of a gas or particulate matter concentration or emission rate.” They are used to determine compliance with EPA emissions standards and are therefore associated with a given “smokestack” and are categorized in the raw data by a corresponding `unitid`. Because point sources of pollution are not alway correlated on a one-to-one basis with generation units, the CEMS `unitid` serves as its own unique grouping. The EPA in collaboration with the EIA has developed a crosswalk table that maps the EPA’s `unitid` onto EIA’s `boiler_id`, `generator_id`, and `plant_id_eia`. This file has been integrated into the SQL database.

The EPA Clean Air Markets Division (CAMD) has collected emissions data from CEMS units stretching back to 1995. Among the data included in CEMS are hourly SO2, CO2, NOx emission and gross load.

Who is required to install CEMS and report to EPA?

Part 75 of the Federal Code of Regulations (FRC), the backbone of the Clean Air Act Title IV and Acid Rain Program, requires coal and other solid-combusting units (see §72.2) to install and use CEMS (see §75.2, §72.6). Certain low-sulfur fueled gas and oil units (see §72.2) may seek exemption or alternative means of monitoring their emissions if desired (see §§75.23, §§75.48, §§75.66). Once CEMS are installed, Part 75 requires hourly data recording, including during startup, shutdown, and instances of malfunction as well as quarterly data reporting to the EPA. The regulation further details the protocol for missing data calculations and backup monitoring for instances of CEMS failure (see §§75,31-37).

A plain English explanation of the requirements of Part 75 is available in section 2.0 Overview of Part 75 Monitoring Requirements

What does the original data look like?

EPA CAMD publishes the CEMS data in an online data portal . The files are available in a prepackaged format, accessible via a user interface or FTP site with each downloadable zip file encompassing a year of data.

How much of the data is accessible through PUDL?

All of it!

Notable Irregularities

CEMS 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 with Dask. Check out the EPA CEMS example notebook in our pudl-examples repository on GitHub for pointers on how to access this big dataset efficiently using `dask`.

PUDL Data Tables

Clicking on the links will show you the names and descriptions of the fields available in the CEMS table.

- *hourly_emissions_epacems*

PUDL Data Transformations

The PUDL transformation process cleans the input data so that it is adjusted for uniformity, corrected for errors, and ready for bulk programmatic use.

To see the transformations applied to the data in each table, you can read the documentation for *`pudl.transform.epacems`* created for their respective transform functions.

Thanks to Karl Dunkle Werner for contributing much of the EPA CEMS Hourly ETL code!

8.3.4 FERC Form 1

Source URL	https://www.ferc.gov/industries-data/electric/general-information/electric-industry-forms/form-1-electric-utility-annual
Source Description	Financial and operational information from electric utilities, licensees and others entities subject to FERC jurisdiction.
Respondents	Major electric utilities and licensees
Source Format	FoxPro Database (.DBC/.DBF)
Source Years	1994-2019
Size (Download)	1.3 GB
PUDL Code	`ferc1`
Years Liberated	1994-2019
Records Liberated	~12 million (116 raw tables), ~316,000 (7 clean tables)
Issues	Open FERC Form 1 issues

Background

The FERC Form 1, otherwise known as the **Electric Utility Annual Report**, contains financial and operating data for major utilities and licensees. Much of it is not publicly available anywhere else.

Who is required to fill out the form?

As outlined in the Commission's Uniform System of Accounts Prescribed for Public Utilities and Licensees Subject To the Provisions of The Federal Power Act (18 C.F.R. Part 101), to qualify as a respondent, entities must exceed at least one of the following criteria for three consecutive years prior to reporting:

- 1 million MWh of total sales
- 100MWh of annual sales for resale
- 500MWh of annual power exchanges delivered
- 500MWh of annual wheeling for others (deliveries plus losses)

Annual responses are due in April of the following year. FERC typically releases the new data in October.

How much of the data is accessible through PUDL?

Thus far we have integrated 7 tables into the full PUDL ETL pipeline. We focused on the tables pertaining to power plants, their capital & operating expenses, and fuel consumption; however, we have the tools required to pull just about any other table in as well.

What does the original data look like?

See also:

Explore the full FERC Form 1 dataset at: https://data.catalyst.coop/ferc1

The data is published as a collection of Visual FoxPro databases, one per year beginning in 1994. The databases all share a very similar structure, with a total of 116 data tables containing ~8GB of raw data (though 90% of that data is in 3 tables containing binary data). The final release of Visual FoxPro was v9.0 in 2007. Its extended support period ended in 2015. The bridge application which allowed this database to be used in Microsoft Access has been discontinued. FERC's continued use of this database format creates a significant barrier to data access.

The FERC 1 database is poorly normalized, and the data itself does not appear to be subject to much quality control. For more detaild context and documentation on a table-by-table basis, see *FERC Form 1 Data Dictionary*

Notable Irregularities

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

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.

Lots of the data included in the FERC tables is extraneous and difficult to parse. None of the tables have record identification, and they sometimes contain multiple rows pertaining to the same plant or portion of a plant. For example, a utility might report values for individual plants as well as the sum total, rendering any aggregations performed on the column inaccurate. Sometimes there are values reported for the total rows and not the individual plants, making them difficult to simply remove. Moreover, these duplicate rows are incredibly difficult to identify.

To improve their usability, we have developed a complex system of regional mapping in order to create ids for each of the plants that can then be compared to PUDL ids and used for integration with EIA and other data. We also remove many of the duplicate rows, and are in the midst of executing a more thorough review of the extraneous rows.

Over time we will pull in and clean up additional FERC Form 1 tables. If there's data you need from Form 1 in bulk you can hire us to liberate it first.

PUDL Data Tables

We've segmented the processed FERC Form 1 data into the following normalized data tables. Clicking on the links will show you the names and descriptions of the fields available in each table.

Data Dictionary	Browse Online
fuel_ferc1	https://data.catalyst.coop/pudl/fuel_ferc1
plant_in_service_ferc1	https://data.catalyst.coop/pudl/plant_in_service_ferc1
plants_ferc1	https://data.catalyst.coop/pudl/plants_ferc1
plants_hydro_ferc1	https://data.catalyst.coop/pudl/plants_hydro_ferc1
plants_pumped_storage_ferc1	https://data.catalyst.coop/pudl/plants_pumped_storage_ferc1
plants_small_ferc1	https://data.catalyst.coop/pudl/plants_small_ferc1
plants_steam_ferc1	https://data.catalyst.coop/pudl/plants_steam_ferc1
purchased_power_ferc1	https://data.catalyst.coop/pudl/purchased_power_ferc1
utilities_ferc1	https://data.catalyst.coop/pudl/utilities_ferc1

PUDL Data Transformations

To see the transformations applied to the data in each table, you can read the `pudl.transform.ferc1` module documentation for more details. created for their respective transform functions.

8.3.5 FERC Form 1 Data Dictionary

We have mapped the Visual FoxPro DBF files to their corresponding FERC Form 1 database tables, and provided a short description of the contents of each table here.

- `A diagram of the 2015 FERC Form 1 Database (PDF)`
- `Blank FERC Form 1 (PDF, to 2014-12-31)`
- `Blank FERC Form 1 (PDF, to 2019-12-31)`
- `Blank FERC Form 1 (PDF, to 2022-11-30)`

Note:

- The Table Names link to the contents of the database table on our FERC Form 1 Datasette deployment, where you can browse and query the raw data yourself, or download the SQLite DB in its entirety.
- The mapping of File Name to Table Name is consistent across all years of data.

- Page numbers correspond to the pages of the FERC Form 1 PDF as it appeared in 2015, and may not be valid for other years.
- Many tables without descriptions were discontinued prior to 2015.
- The "Freq" column indicates the reporting frequency – A for Annual; Q for Quarterly. A/Q if the data is reported both annually and quarterly.

Table Name / Data Link	File Name	Pages	Freq	Table Description
f1_106_2009	F1_106_2009.DBF	106	A	Information on Formula Rates
f1_106a_2009	F1_106A_2009.DBF	106	A	Information on Formula Rates
f1_106b_2009	F1_106B_2009.DBF	106	A	Information on Formula Rates
f1_208_elc_dep	F1_208_ELC_DEP.DBF	208	Q	Electric Plant In Service and Accumulated
f1_231_trn_stdycst	F1_231_TRN_STDYCST.DBF	231	A/Q	Transmission Service and Generation Interc
f1_324_elc_expns	F1_324_ELC_EXPNS.DBF	324	Q	Electric Production, Other Power Supply E
f1_325_elc_cust	F1_325_ELC_CUST.DBF	325	Q	Electric Customer Accounts, Service, Sales
f1_331_transiso	F1_331_TRANSISO.DBF	331	A/Q	Transmission of Electricity by ISO/RTOs
f1_338_dep_depl	F1_338_DEP_DEPL.DBF	338	Q	Depreciation, Depletion and Amortization o
f1_397_isorto_stl	F1_397_ISORTO_STL.DBF	397	A/Q	Amounts Included in ISO/RTO Settlement
f1_398_ancl_ps	F1_398_ANCL_PS.DBF	398	A	Purchases and Sales of Ancillary Services
f1_399_mth_peak	F1_399_MTH_PEAK.DBF	399	A/Q	Monthly Peak Loads and Energy Output
f1_400_sys_peak	F1_400_SYS_PEAK.DBF	400	A/Q	Monthly Transmission System Peak Load
f1_400a_iso_peak	F1_400A_ISO_PEAK.DBF	980, 400a	A/Q	Monthly ISO/RTO Transmission System Pe
f1_429_trans_aff	F1_429_TRANS_AFF.DBF	429	A	Transactions with Associated (Affiliated) C
f1_acb_epda	F1_2.DBF	336-337	A	Depreciation & Amortization of Electric Pl
f1_accumdepr_prvsn	F1_3.DBF	219	A	Accumulated Provision for Depreciation of
f1_accumdfrrdtaxcr	F1_4.DBF	266-267	A	Accumulated Deferred Investment Tax Cred
f1_adit_190_detail	F1_5.DBF	234-234a	A	Accumulated Deferred Income Taxes (Indiv
f1_adit_190_notes	F1_6.DBF	234-234b	A	Accumulated Deferred Income Taxes (Note
f1_adit_amrt_prop	F1_7.DBF	272-273	A	Accumulated Deferred Income Taxes - Acc
f1_adit_other	F1_8.DBF	276-277	A	Accumulated Deferred Income Taxes - Othe
f1_adit_other_prop	F1_9.DBF	274-275	A	Accumulated Deferred Income Taxes - Othe
f1_allowances	F1_10.DBF	228-229	A	Allowances
f1_allowances_nox	F1_ALLOWANCES_NOX.DBF	230-230a	A	
f1_audit_log	F1_78.DBF			
f1_bal_sheet_cr	F1_11.DBF	112-113	A/Q	Comparative Balance Sheet (Liabilities & C
f1_capital_stock	F1_12.DBF	250-251	A	Capital Stock
f1_cash_flow	F1_13.DBF	120-121	A/Q	Statement of Cash Flows
f1_cmmn_utlty_p_e	F1_14.DBF	356	A	Common Utility Plant & Expenses
f1_cmpinc_hedge	F1_CMPINC_HEDGE.DBF	990, 122(a)(b)	A/Q	Statement of Accumulated Comparative Inc
f1_cmpinc_hedge_a	F1_CMPINC_HEDGE_A.DBF	990		
f1_co_directors	F1_18.DBF	105	A	Names, Titles, and Addresses of Directors
f1_codes_val	F1_76.DBF			
f1_col_lit_tbl	F1_79.DBF			Descriptive headers for each column in the
f1_comp_balance_db	F1_15.DBF	110-111	A/Q	Comparative Balance Sheet (Assets & Othe
f1_construction	F1_16.DBF	217		Spending on Construction (1994-2002 only
f1_control_respdnt	F1_17.DBF	102	A	Control Over Respondent
f1_cptl_stk_expns	F1_19.DBF	254-254b	A	Capital Stock Expense
f1_csscslc_pcsircs	F1_20.DBF	252		
f1_dacs_epda	F1_21.DBF	336-337	A	Depreciation & Amortization of Electric Pl
f1_dscnt_cptl_stk	F1_22.DBF	254		

Table 1 – continued from previ

Table Name / Data Link	File Name	Pages	Freq	Table Description
f1_edcfu_epda	F1_23.DBF	336-337	A	Depreciation & Amortization of Electric Pl
f1_elc_op_mnt_expn	F1_27.DBF	320-323	A	Electric Operation & Maintenance Expense
f1_elc_oper_rev_nb	F1_26.DBF	300-301b	A/Q	Electric Operating Revenues (Unbilled Rev
f1_elctrc_erg_acct	F1_24.DBF	401-401a	A	Electric Energy Account
f1_elctrc_oper_rev	F1_25.DBF	300-301a	A/Q	Electric Operating Revenues (Individual Sc
f1_electric	F1_28.DBF	429		
f1_email	F1_EMAIL.DBF			
f1_envrnmntl_expns	F1_29.DBF	431		
f1_envrnmntl_fclty	F1_30.DBF	430		
f1_footnote_data	F1_85.DBF	450	A/Q	Footnote Data
f1_footnote_tbl	F1_87.DBF			
f1_freeze	F1_FREEZE.DBF			
f1_fuel	F1_31.DBF	402-403b	A	Steam-Electric Generation Plant Statistics -
f1_general_info	F1_32.DBF	101	A	General Information
f1_gnrt_plant	F1_33.DBF	410-411	A	Generating Plant Statistics (Small Plants)
f1_hydro	F1_86.DBF	406-407	A	Hydroelectric Gen Plant Stats (Large Plants
f1_ident_attsttn	F1_88.DBF	1	A/Q	Identification & Attestation
f1_important_chg	F1_34.DBF	108-109	A/Q	Important Changes During the Quarter/Yea
f1_incm_stmnt_2	F1_35.DBF	114-117b	A/Q	Statement of Income (Other Income & Ded
f1_income_stmnt	F1_36.DBF	114-117a	A/Q	Statement of Income
f1_leased	F1_90.DBF	213	A	Electric Plant Leased to Others
f1_load_file_names	F1_80.DBF			
f1_long_term_debt	F1_93.DBF	256-257	A	Long-Term Debt
f1_misc_dfrrd_dr	F1_38.DBF	233	A	Miscellaneous Deferred Debits
f1_miscgen_expnelc	F1_37.DBF	335	A	Miscellaneous General Expenses - Electric
f1_mthly_peak_otpt	F1_39.DBF	401-401b	A	Monthly Peaks & Output
f1_mtrl_spply	F1_40.DBF	227, 228-229	A	Materials & Supplies
f1_nbr_elc_deptemp	F1_41.DBF	320		
f1_nonutility_prop	F1_42.DBF	221		
f1_note_fin_stmnt	F1_43.DBF	122-123	A/Q	Notes to Financial Statements
f1_nuclear_fuel	F1_44.DBF	202-203	A	Nuclear Fuel Materials
f1_officers_co	F1_45.DBF	104	A	Officers
f1_othr_dfrrd_cr	F1_46.DBF	269	A	Other Deferred Credits
f1_othr_pd_in_cptl	F1_47.DBF	253	A	Other Paid-in Capital
f1_othr_reg_assets	F1_48.DBF	232	A/Q	Other Regulatory Assets
f1_othr_reg_liab	F1_49.DBF	278	A/Q	Other Regulatory Liabilities
f1_overhead	F1_50.DBF	218		
f1_pccidica	F1_51.DBF	340		
f1_pins	F1_PINS.DBF			
f1_plant	F1_92.DBF	204, 214	A	Electric Plant Held for Future Use
f1_plant_in_srvce	F1_52.DBF	204-207	A	Electric Plant in Service
f1_privilege	F1_81.DBF			
f1_pumped_storage	F1_53.DBF	408-409	A	Pumped Storage Generating Plant Statistics
f1_purchased_pwr	F1_54.DBF	326-327	A	Purchased Power
f1_r_d_demo_actvty	F1_59.DBF	352-353	A	Research, Development & Demonstration A
f1_reconrpt_netinc	F1_55.DBF	261	A	Reconciliation of Reported Net Income wit
f1_reg_comm_expn	F1_56.DBF	350-351	A	Regulatory Commission Expenses
f1_respdnt_control	F1_57.DBF	103	A	Corporations Controlled by Respondent
f1_respondent_id	F1_1.DBF			Respondent ID

Table 1 – continued from previ

Table Name / Data Link	File Name	Pages	Freq	Table Description
f1_retained_erng	F1_58.DBF	118-119	A/Q	Statement of Retained Earnings for the Yea
f1_rg_trn_srv_rev	F1_RG_TRN_SRV_REV.DBF	302	A/Q	Regional Transmission Service Revenues (A
f1_row_lit_tbl	F1_84.DBF			Descriptive labels for each numbered row i
f1_s0_checks	F1_S0_CHECKS.DBF			
f1_s0_filing_log	F1_S0_FILING_LOG.DBF			
f1_sale_for_resale	F1_61.DBF	310-311	A	Sales for Resale
f1_sales_by_sched	F1_60.DBF	304	A	Sales of Electricity by Rate Schedules
f1_sbsdry_detail	F1_91.DBF	224-225	A	Investment in Subsidiary Companies (Acco
f1_sbsdry_totals	F1_62.DBF	224-225	A	Investment in Subsidiary Companies (Total
f1_sched_lit_tbl	F1_77.DBF			
f1_schedules_list	F1_63.DBF	002-004	A/Q	List of Schedules
f1_security	F1_SECURITY.DBF	106		
f1_security_holder	F1_64.DBF	106		
f1_slry_wg_dstrbtn	F1_65.DBF	354-355	A	Distribution of Salaries & Wages
f1_steam	F1_89.DBF	402-403a	A	Steam-Electric Generation Plant Statistics -
f1_substations	F1_66.DBF	426-427	A	Substations
f1_sys_error_log	F1_82.DBF			
f1_taxacc_ppchrgyr	F1_67.DBF	262-263	A	Taxes Accrued, Prepaid & Charged During
f1_unique_num_val	F1_83.DBF			
f1_unrcvrd_cost	F1_68.DBF	230-230b	A	Unrecovered Plant & Regulatory Study Cos
f1_utltyplnt_smmry	F1_69.DBF	200-201	A/Q	Summary of Utility Plant & Accumulated P
f1_work	F1_70.DBF	216	A	Construction Work in Progress - Electric
f1_xmssn_adds	F1_71.DBF	424-425	A	Transmission Lines Added During Year
f1_xmssn_elc_bothr	F1_72.DBF	332	A/Q	Transmission of Electricity by Others
f1_xmssn_elc_fothr	F1_73.DBF	328-330	A/Q	Transmission of Electricity for Others
f1_xmssn_line	F1_74.DBF	422-423	A	Transmission Line Statistics
f1_xtraordnry_loss	F1_75.DBF	230-230a	A	Extraordinary Property Losses

8.3.6 Work in Progress & Future Datasets

Contents

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 between April, 2021 and March 2023.

There's a huge variety and quantity of data about the US electric utility system available to the public. The data we have integrated 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.

Census DP1

The US Census Demographic Profile 1 (DP1) provides Census tract, county, and state-level demographic information, along with the geometries defining those areas. We use this information in generating historical utility and balancing authority service territories based on FERC 714 and EIA 861 data. Currently we are distributing the Census DP1 data as a standalone SQLite DB.

EIA Form 861

The EIA Form 861, also known as the **Annual Electric Power Industry Report**, compiles information on load, generation, capacity, sales, revenues, programs, and more. Right now we've got all of 861 integrated and are building out our testing and data validation before publishing the data officially.

- `EIA-861 Instructions (PDF, to 2020-03-31)`
- `EIA-861 Instructions (PDF, to 2023-05-31)`

EIA Form 176

EIA Form 176, also known as the Annual Report of Natural and Supplemental Gas Supply and Disposition, describes the origins, suppliers, and disposition of natural gas on a yearly and state by state basis.

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 CSV files exported from a Visual FoxPro database prior to publication. All of the raw tables are being extracted, and a couple of them have been integrated into the transform process. None are in the PUDL DB yet.

- `FERC-714 Instructions (PDF, as of 2021-04-16)`

FERC EQR

The FERC Electric Quarterly Reports (EQR), also known FERC 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.

FERC Form 2

FERC Form 2 is analogous to FERC Form 1, but pertains to gas rather than electric utilities. It paints a detailed picture of the finances of natural gas utilities.

PHMSA Natural Gas Pipelines

The PHMSA Natural Gas Annual Report, published by the Pipeline and Hazardous Materials Safety Administration (which is part of the US Dept. of Transportation) collects data about the natural gas gathering, transmission and distribution system, including their age, length, diameter, materials, and carrying capacity. It includes information about natural gas storage facilities and liquefied natural gas shipping facilities as well.

Machine Readable Clean Energy Standards

Renewable Portfolio Standards (RPS) and Clean Energy Standards (CES) have emerged as one of the primary policy tools to decarbonize the US electricity supply. Researchers who model future electricity systems need to include these binding regulations as constraints on their models to ensure that the systems they explore are legally compliant. Unfortunately for modelers, RPS and CES regulations vary from state to state. Sometimes there are carve outs for different types of generation, and sometimes there are different requirements for different types of utilities or distributed resources. Our goal is to compile a programmatically usable database of RPS/CES policies in the US for quick and easy reference by modelers.

Future Data of Interest

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!).

EIA Water Usage

EIA Water records water use by thermal generating stations in the US.

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.

8.4 PUDL Data Dictionary

8.4.1 assn_gen_eia_unit_epa

Browse or query this table in Datasette.

Field Name	Type	Description
generator_id	string	Generator identification code. Often numeric, but sometimes includes letters. It's a string!
plant_id_eia	integer	The unique six-digit facility identification number, also called an ORISPL, assigned by the Energy Information Administration.
unit_id_epa	string	Smokestack unit monitored by EPA CEMS.

8.4.2 assn_plant_id_eia_epa

Browse or query this table in Datasette.

Field Name	Type	Description
plant_id_eia	integer	The unique six-digit facility identification number, also called an ORISPL, assigned by the Energy Information Administration.
plant_id_epa	integer	N/A

8.4.3 boiler_fuel_eia923

Browse or query this table in Datasette.

Field Name	Type	Description
ash_content_pct	number	Ash content percentage by weight to the nearest 0.1 percent.
boiler_id	string	Boiler identification code. Alphanumeric.
fuel_consumed_units	number	Consumption of the fuel type in physical units. Note: this is the total quantity consumed for both electricity and, in the case of combined heat and power plants, process steam production.
fuel_mmbtu_per_unit	number	Heat content of the fuel in millions of Btus per physical unit.
fuel_type_code	string	The fuel code reported to EIA. Two or three letter alphanumeric.
fuel_type_code_pudl	string	Standardized fuel codes in PUDL.
plant_id_eia	integer	The unique six-digit facility identification number, also called an ORISPL, assigned by the Energy Information Administration.
report_date	date	Date reported.
sulfur_content_pct	number	Sulfur content percentage by weight to the nearest 0.01 percent.

8.4.4 boiler_generator_assn_eia860

Browse or query this table in Datasette.

Field Name	Type	Description
bga_source	string	The source from where the unit_id_pudl is compiled. The unit_id_pudl comes directly from EIA 860, or string association (which looks at all the boilers and generators that are not associated with a unit and tries to find a matching string in the respective collection of boilers or generator), or from a unit connection (where the unit_id_eia is employed to find additional boiler generator connections).
boiler_id	string	EIA-assigned boiler identification code.
generator_id	string	EIA-assigned generator identification code.
plant_id_eia	integer	The unique six-digit facility identification number, also called an ORISPL, assigned by the Energy Information Administration.
report_date	date	Date reported.
unit_id_eia	string	EIA-assigned unit identification code.
unit_id_pudl	integer	PUDL-assigned unit identification number.

8.4.5 boilers_entity_eia

Browse or query this table in Datasette.

Field Name	Type	Description
boiler_id	string	The EIA-assigned boiler identification code. Alphanumeric.
plant_id_eia	integer	The unique six-digit facility identification number, also called an ORISPL, assigned by the Energy Information Administration.
prime_mover_code	string	Code for the type of prime mover (e.g. CT, CG)

8.4.6 coalmine_eia923

Browse or query this table in Datasette.

Field Name	Type	Description
county_id_fips	integer	County ID from the Federal Information Processing Standard Publication 6-4.
mine_id_msha	integer	MSHA issued mine identifier.
mine_id_pudl	integer	PUDL issued surrogate key.
mine_name	string	Coal mine name.
mine_type_code	string	Type of mine. P: Preparation plant, U: Underground, S: Surface, SU: Mostly Surface with some Underground, US: Mostly Underground with some Surface.
state	string	Two letter US state abbreviations and three letter ISO-3166-1 country codes for international mines.

8.4.7 energy_source_eia923

Browse or query this table in Datasette.

Field Name	Type	Description
abbr	string	N/A
source	string	N/A

8.4.8 ferc_accounts

Browse or query this table in Datasette.

Field Name	Type	Description
description	string	Long description of the FERC Account.
ferc_account_id	string	Account number, from FERC's Uniform System of Accounts for Electric Plant. Also includes higher level labeled categories.

8.4.9 ferc_depreciation_lines

Browse or query this table in Datasette.

Field Name	Type	Description
description	string	Description of the FERC depreciation account, as listed on FERC Form 1, Page 219.
line_id	string	A human readable string uniquely identifying the FERC depreciation account. Used in lieu of the actual line number, as those numbers are not guaranteed to be consistent from year to year.

8.4.10 fuel_ferc1

Browse or query this table in Datasette.

Field Name	Type	Description
fuel_cost_per_mmbtu	number	Average cost of fuel consumed in the report year, in nominal USD per mmBTU of fuel heat content.
fuel_cost_per_unit_burned	number	Average cost of fuel consumed in the report year, in nominal USD per reported fuel unit.
fuel_cost_per_unit_delivered	number	Average cost of fuel delivered in the report year, in nominal USD per reported fuel unit.
fuel_mmbtu_per_unit	number	Average heat content of fuel consumed in the report year, in mmBTU per reported fuel unit.
fuel_qty_burned	number	Quantity of fuel consumed in the report year, in terms of the reported fuel units.
fuel_type_code_pudl	string	PUDL assigned code indicating the general fuel type.
fuel_unit	string	PUDL assigned code indicating reported fuel unit of measure.
plant_name_ferc1	string	Name of the plant, as reported to FERC. This is a freeform string, not guaranteed to be consistent across references to the same plant.
record_id	string	Identifier indicating original FERC Form 1 source record. format: {table_name}_{report_year}_{report_prd}_{respondent_id}_{spplmnt_num}_{row_number}. Unique within FERC Form 1 DB tables which are not row-mapped.
report_year	year	Four-digit year in which the data was reported.
utility_id_ferc1	integer	FERC assigned respondent_id, identifying the reporting entity. Stable from year to year.

8.4.11 fuel_receipts_costs_eia923

Browse or query this table in Datasette.

Field Name	Type	Description
ash_content_pct	number	Ash content percentage by weight to the nearest 0.1 percent.
chlorine_content_ppm	number	N/A
contract_expiration_date	date	Date contract expires.Format: MMYY.
contract_type_code	string	Purchase type under which receipts occurred in the reporting month. C: Contract, NC: New Contract, S: Spot Purchase, T: Tolling Agreement.
energy_source_code	string	The fuel code associated with the fuel receipt. Two or three character alphanumeric.
fuel_cost_per_mmbtu	number	All costs incurred in the purchase and delivery of the fuel to the plant in cents per million Btu(MMBtu) to the nearest 0.1 cent.
fuel_group_code	string	Groups the energy sources into fuel groups that are located in the Electric Power Monthly: Coal, Natural Gas, Petroleum, Petroleum Coke.
fuel_group_code_simple	string	Simplified grouping of fuel_group_code, with Coal and Petroluem Coke as well as Natural Gas and Other Gas grouped together.
fuel_qty_units	number	Quanity of fuel received in tons, barrel, or Mcf.
fuel_type_code_pudl	string	Standardized fuel codes in PUDL.
heat_content_mmbtu_per_unit	number	Heat content of the fuel in millions of Btus per physical unit to the nearest 0.01 percent.
id	integer	PUDL issued surrogate key.
mercury_content_ppm	number	Mercury content in parts per million (ppm) to the nearest 0.001 ppm.
mine_id_pudl	integer	PUDL mine identification number.
moisture_content_pct	number	N/A
natural_gas_delivery_contract_type_code	string	Contract type for natrual gas delivery service:
natural_gas_transport_code	string	Contract type for natural gas transportation service.
plant_id_eia	integer	The unique six-digit facility identification number, also called an ORISPL, assigned by the Energy Information Administration.
primary_transportation_mode_code	string	Transportation mode for the longest distance transported.
report_date	date	Date reported.
secondary_transportation_mode_code	string	Transportation mode for the second longest distance transported.
sulfur_content_pct	number	Sulfur content percentage by weight to the nearest 0.01 percent.
supplier_name	string	Company that sold the fuel to the plant or, in the case of Natural Gas, pipline owner.

8.4.12 fuel_type_aer_eia923

Browse or query this table in Datasette.

Field Name	Type	Description
abbr	string	N/A
fuel_type	string	N/A

8.4.13 fuel_type_eia923

Browse or query this table in Datasette.

Field Name	Type	Description
abbr	string	N/A
fuel_type	string	N/A

8.4.14 generation_eia923

Browse or query this table in Datasette.

Field Name	Type	Description
generator_id	string	Generator identification code. Often numeric, but sometimes includes letters. It's a string!
net_generation_mwh	number	Net generation for specified period in megawatthours (MWh).
plant_id_eia	integer	The unique six-digit facility identification number, also called an ORISPL, assigned by the Energy Information Administration.
report_date	date	Date reported.

8.4.15 generation_fuel_eia923

Browse or query this table in Datasette.

Field Name	Type	Description
fuel_consumed_for_electricity_mmbtu	number	Total consumption of fuel to produce electricity, in physical units, year to date.
fuel_consumed_for_electricity_units	number	Consumption for electric generation of the fuel type in physical units.
fuel_consumed_mmbtu	number	Total consumption of fuel in physical units, year to date. Note: this is the total quantity consumed for both electricity and, in the case of combined heat and power plants, process steam production.
fuel_consumed_units	number	Consumption of the fuel type in physical units. Note: this is the total quantity consumed for both electricity and, in the case of combined heat and power plants, process steam production.
fuel_mmbtu_per_unit	number	Heat content of the fuel in millions of Btus per physical unit.
fuel_type	string	The fuel code reported to EIA. Two or three letter alphanumeric.
fuel_type_code_aer	string	A partial aggregation of the reported fuel type codes into larger categories used by EIA in, for example, the Annual Energy Review (AER).Two or three letter alphanumeric.
fuel_type_code_pudl	string	Standardized fuel codes in PUDL.
net_generation_mwh	number	Net generation, year to date in megawatthours (MWh). This is total electrical output net of station service. In the case of combined heat and power plants, this value is intended to include internal consumption of electricity for the purposes of a production process, as well as power put on the grid.
nuclear_unit_id	integer	For nuclear plants only, the unit number .One digit numeric. Nuclear plants are the only type of plants for which data are shown explicitly at the generating unit level.
plant_id_eia	integer	The unique six-digit facility identification number, also called an ORISPL, assigned by the Energy Information Administration.
prime_mover_code	string	Type of prime mover.
report_date	date	Date reported.

8.4.16 generators_eia860

Browse or query this table in Datasette.

Field Name	Type	Description
capacity_mw	number	The highest value on the generator nameplate in megawatts rounded to the n
carbon_capture	boolean	Indicates whether the generator uses carbon capture technology.
cofire_fuels	boolean	Can the generator co-fire fuels?.
current_planned_operating_date	date	The most recently updated effective date on which the generator is schedule
data_source	string	Source of EIA 860 data. Either Annual EIA 860 or the year-to-date updates
deliver_power_transgrid	boolean	Indicate whether the generator can deliver power to the transmission grid.
distributed_generation	boolean	Whether the generator is considered distributed generation
energy_source_1_transport_1	string	Primary Mode of Transportaion for Energy Source 1
energy_source_1_transport_2	string	Secondary Mode of Transportaion for Energy Source 1
energy_source_1_transport_3	string	Third Mode of Transportaion for Energy Source 1
energy_source_2_transport_1	string	Primary Mode of Transportaion for Energy Source 2

Table 2 – continued from previous page

Field Name	Type	Description
energy_source_2_transport_2	string	Secondary Mode of Transportaion for Energy Source 2
energy_source_2_transport_3	string	Third Mode of Transportaion for Energy Source 2
energy_source_code_1	string	The code representing the most predominant type of energy that fuels the ge
energy_source_code_2	string	The code representing the second most predominant type of energy that fuel
energy_source_code_3	string	The code representing the third most predominant type of energy that fuels t
energy_source_code_4	string	The code representing the fourth most predominant type of energy that fuels
energy_source_code_5	string	The code representing the fifth most predominant type of energy that fuels th
energy_source_code_6	string	The code representing the sixth most predominant type of energy that fuels t
fuel_type_code_pudl	string	Standardized fuel codes in PUDL.
generator_id	string	Generator identification number.
minimum_load_mw	number	The minimum load at which the generator can operate at continuosuly.
multiple_fuels	boolean	Can the generator burn multiple fuels?
nameplate_power_factor	number	The nameplate power factor of the generator.
operational_status	string	The operating status of the generator. This is based on which tab the generat
operational_status_code	string	The operating status of the generator.
other_modifications_date	date	Planned effective date that the generator is scheduled to enter commercial op
other_planned_modifications	boolean	Indicates whether there are there other modifications planned for the generat
owned_by_non_utility	boolean	Whether any part of generator is owned by a nonutilty
ownership_code	string	Identifies the ownership for each generator.
planned_derate_date	date	Planned effective month that the generator is scheduled to enter operation af
planned_energy_source_code_1	string	New energy source code for the planned repowered generator.
planned_modifications	boolean	Indicates whether there are any planned capacity uprates/derates, repowering
planned_net_summer_capacity_derate_mw	number	Decrease in summer capacity expected to be realized from the derate modifi
planned_net_summer_capacity_uprate_mw	number	Increase in summer capacity expected to be realized from the modification to
planned_net_winter_capacity_derate_mw	number	Decrease in winter capacity expected to be realized from the derate modifica
planned_net_winter_capacity_uprate_mw	number	Increase in winter capacity expected to be realized from the uprate modificat
planned_new_capacity_mw	number	The expected new namplate capacity for the generator.
planned_new_prime_mover_code	string	New prime mover for the planned repowered generator.
planned_repower_date	date	Planned effective date that the generator is scheduled to enter operation after
planned_retirement_date	date	Planned effective date of the scheduled retirement of the generator.
planned_uprate_date	date	Planned effective date that the generator is scheduled to enter operation after
plant_id_eia	integer	The unique six-digit facility identification number, also called an ORISPL, a
reactive_power_output_mvar	number	Reactive Power Output (MVAr)
report_date	date	Date reported.
retirement_date	date	Date of the scheduled or effected retirement of the generator.
startup_source_code_1	string	The code representing the first, second, third or fourth start-up and flame sta
startup_source_code_2	string	The code representing the first, second, third or fourth start-up and flame sta
startup_source_code_3	string	The code representing the first, second, third or fourth start-up and flame sta
startup_source_code_4	string	The code representing the first, second, third or fourth start-up and flame sta
summer_capacity_estimate	boolean	Whether the summer capacity value was an estimate
summer_capacity_mw	number	The net summer capacity.
summer_estimated_capability_mw	number	EIA estimated summer capacity (in MWh).
switch_oil_gas	boolean	Indicates whether the generator switch between oil and natural gas.
syncronized_transmission_grid	boolean	Indicates whether standby generators (SB status) can be synchronized to the
technology_description	string	High level description of the technology used by the generator to produce ele
time_cold_shutdown_full_load_code	string	The minimum amount of time required to bring the unit to full load from shu
turbines_inverters_hydrokinetics	string	Number of wind turbines, or hydrokinetic buoys.
turbines_num	integer	Number of wind turbines, or hydrokinetic buoys.
uprate_derate_completed_date	date	The date when the uprate or derate was completed.

Table 2 – continued from previous page

Field Name	Type	Description
uprate_derate_during_year	boolean	Was an uprate or derate completed on this generator during the reporting yea
utility_id_eia	integer	EIA-assigned identification number for the company that is responsible for t
winter_capacity_estimate	boolean	Whether the winter capacity value was an estimate
winter_capacity_mw	number	The net winter capacity.
winter_estimated_capability_mw	number	EIA estimated winter capacity (in MWh).

8.4.17 generators_entity_eia

Browse or query this table in Datasette.

Field Name	Type	Description
associated_combined_heat_power	boolean	Indicates whether the generator is associated with a combined heat and power system
bypass_heat_recovery	boolean	Can this generator operate while bypassing the heat recovery steam generator?
duct_burners	boolean	Indicates whether the unit has duct-burners for supplementary firing of the turbine exhaust gas
fluidized_bed_tech	boolean	Indicates whether the generator uses fluidized bed technology
generator_id	string	Generator identification number
operating_date	date	Date the generator began commercial operation
operating_switch	string	Indicates whether the fuel switching generator can switch when operating
original_planned_operating_date	date	The date the generator was originally scheduled to be operational
other_combustion_tech	boolean	Indicates whether the generator uses other combustion technologies
plant_id_eia	integer	The unique six-digit facility identification number, also called an ORISPL, assigned by the Energy Information Administration.
previously_canceled	boolean	Indicates whether the generator was previously reported as indefinitely postponed or canceled
prime_mover_code	string	EIA assigned code for the prime mover (i.e. the engine, turbine, water wheel, or similar machine that drives an electric generator)
pulverized_coal_tech	boolean	Indicates whether the generator uses pulverized coal technology
rto_iso_lmp_node_id	string	The designation used to identify the price node in RTO/ISO Locational Marginal Price reports
rto_iso_location_wholesale_reporting_id	string	The designation used to report ths specific location of the wholesale sales transactions to FERC for the Electric Quarterly Report
solid_fuel_gasification	boolean	Indicates whether the generator is part of a solid fuel gasification system
stoker_tech	boolean	Indicates whether the generator uses stoker technology
subcritical_tech	boolean	Indicates whether the generator uses subcritical technology
supercritical_tech	boolean	Indicates whether the generator uses supercritical technology
topping_bottoming_code	string	If the generator is associated with a combined heat and power system, indicates whether the generator is part of a topping cycle or a bottoming cycle
ultrasupercritical_tech	boolean	Indicates whether the generator uses ultra-supercritical technology

8.4.18 hourly_emissions_epacems

Browse or query this table in Datasette.

Field Name	Type	Description
co2_mass_measurement_code	string	Identifies whether the reported value of emissions was measured, calculated, or measured and substitute.
co2_mass_tons	number	Carbon dioxide emissions in short tons.
facility_id	integer	New EPA plant ID.
gross_load_mw	number	Average power in megawatts delivered during time interval measured.
heat_content_mmbtu	number	The energy contained in fuel burned, measured in million BTU.
nox_mass_lbs	number	NOx emissions in pounds.
nox_mass_measurement_code	string	Identifies whether the reported value of emissions was measured, calculated, or measured and substitute.
nox_rate_lbs_mmbtu	number	The average rate at which NOx was emitted during a given time period.
nox_rate_measurement_code	string	Identifies whether the reported value of emissions was measured, calculated, or measured and substitute.
operating_datetime_utc	datetime	Date and time measurement began (UTC).
operating_time_hours	number	Length of time interval measured.
plant_id_eia	integer	The unique six-digit facility identification number, also called an ORISPL, assigned by the Energy Information Administration.
so2_mass_lbs	number	Sulfur dioxide emissions in pounds.
so2_mass_measurement_code	string	Identifies whether the reported value of emissions was measured, calculated, or measured and substitute.
state	string	State the plant is located in.
steam_load_1000_lbs	number	Total steam pressure produced by a unit during the reported hour.
unit_id_epa	integer	Smokestack unit monitored by EPA CEMS.
unitid	string	Facility-specific unit id (e.g. Unit 4)

8.4.19 ownership_eia860

Browse or query this table in Datasette.

Field Name	Type	Description
fraction_owned	number	Proportion of generator ownership.
generator_id	string	Generator identification number.
owner_city	string	City of owner.
owner_name	string	Name of owner.
owner_state	string	Two letter US & Canadian state and territory abbreviations.
owner_street_address	string	Steet address of owner.
owner_utility_id_eia	integer	EIA-assigned owner's identification number.
owner_zip_code	string	Zip code of owner.
plant_id_eia	integer	The unique six-digit facility identification number, also called an ORISPL, assigned by the Energy Information Administration.
report_date	date	Date reported.
utility_id_eia	integer	EIA-assigned identification number for the company that is responsible for the day-to-day operations of the generator.

8.4.20 plant_in_service_ferc1

Browse or query this table in Datasette.

Field Name	Type	Description
amount_type	string	String indicating which original FERC Form 1 column the listed amount
distribution_acct360_land	number	FERC Account 360: Distribution Plant Land and Land Rights.
distribution_acct361_structures	number	FERC Account 361: Distribution Plant Structures and Improvements.
distribution_acct362_station_equip	number	FERC Account 362: Distribution Plant Station Equipment.
distribution_acct363_storage_battery_equip	number	FERC Account 363: Distribution Plant Storage Battery Equipment.
distribution_acct364_poles_towers	number	FERC Account 364: Distribution Plant Poles, Towers, and Fixtures.
distribution_acct365_overhead_conductors	number	FERC Account 365: Distribution Plant Overhead Conductors and Devic
distribution_acct366_underground_conduit	number	FERC Account 366: Distribution Plant Underground Conduit.
distribution_acct367_underground_conductors	number	FERC Account 367: Distribution Plant Underground Conductors and De
distribution_acct368_line_transformers	number	FERC Account 368: Distribution Plant Line Transformers.
distribution_acct369_services	number	FERC Account 369: Distribution Plant Services.
distribution_acct370_meters	number	FERC Account 370: Distribution Plant Meters.
distribution_acct371_customer_installations	number	FERC Account 371: Distribution Plant Installations on Customer Premis
distribution_acct372_leased_property	number	FERC Account 372: Distribution Plant Leased Property on Customer Pr
distribution_acct373_street_lighting	number	FERC Account 373: Distribution PLant Street Lighting and Signal Syste
distribution_acct374_asset_retirement	number	FERC Account 374: Distribution Plant Asset Retirement Costs.
distribution_total	number	Distribution Plant Total (FERC Accounts 360-374).
electric_plant_in_service_total	number	Total Electric Plant in Service (FERC Accounts 101, 102, 103 and 106)
electric_plant_purchased_acct102	number	FERC Account 102: Electric Plant Purchased.
electric_plant_sold_acct102	number	FERC Account 102: Electric Plant Sold (Negative).
experimental_plant_acct103	number	FERC Account 103: Experimental Plant Unclassified.
general_acct389_land	number	FERC Account 389: General Land and Land Rights.
general_acct390_structures	number	FERC Account 390: General Structures and Improvements.
general_acct391_office_equip	number	FERC Account 391: General Office Furniture and Equipment.

Field Name	Type	Description
general_acct392_transportation_equip	number	FERC Account 392: General Transportation Equipment.
general_acct393_stores_equip	number	FERC Account 393: General Stores Equipment.
general_acct394_shop_equip	number	FERC Account 394: General Tools, Shop, and Garage Equipment.
general_acct395_lab_equip	number	FERC Account 395: General Laboratory Equipment.
general_acct396_power_operated_equip	number	FERC Account 396: General Power Operated Equipment.
general_acct397_communication_equip	number	FERC Account 397: General Communication Equipment.
general_acct398_misc_equip	number	FERC Account 398: General Miscellaneous Equipment.
general_acct399_1_asset_retirement	number	FERC Account 399.1: Asset Retirement Costs for General Plant.
general_acct399_other_property	number	FERC Account 399: General Plant Other Tangible Property.
general_subtotal	number	General Plant Subtotal (FERC Accounts 389-398).
general_total	number	General Plant Total (FERC Accounts 389-399.1).
hydro_acct330_land	number	FERC Account 330: Hydro Land and Land Rights.
hydro_acct331_structures	number	FERC Account 331: Hydro Structures and Improvements.
hydro_acct332_reservoirs_dams_waterways	number	FERC Account 332: Hydro Reservoirs, Dams, and Waterways.
hydro_acct333_wheels_turbines_generators	number	FERC Account 333: Hydro Water Wheels, Turbins, and Generators.
hydro_acct334_accessory_equip	number	FERC Account 334: Hydro Accessory Electric Equipment.
hydro_acct335_misc_equip	number	FERC Account 335: Hydro Miscellaneous Power Plant Equipment.
hydro_acct336_roads_railroads_bridges	number	FERC Account 336: Hydro Roads, Railroads, and Bridges.
hydro_acct337_asset_retirement	number	FERC Account 337: Asset Retirement Costs for Hydraulic Production.
hydro_total	number	Hydraulic Production Plant Total (FERC Accounts 330-337)
intangible_acct301_organization	number	FERC Account 301: Intangible Plant Organization.
intangible_acct302_franchises_consents	number	FERC Account 302: Intangible Plant Franchises and Consents.
intangible_acct303_misc	number	FERC Account 303: Miscellaneous Intangible Plant.
intangible_total	number	Intangible Plant Total (FERC Accounts 301-303).
major_electric_plant_acct101_acct106_total	number	Total Major Electric Plant in Service (FERC Accounts 101 and 106).
nuclear_acct320_land	number	FERC Account 320: Nuclear Land and Land Rights.
nuclear_acct321_structures	number	FERC Account 321: Nuclear Structures and Improvements.
nuclear_acct322_reactor_equip	number	FERC Account 322: Nuclear Reactor Plant Equipment.
nuclear_acct323_turbogenerators	number	FERC Account 323: Nuclear Turbogenerator Units
nuclear_acct324_accessory_equip	number	FERC Account 324: Nuclear Accessory Electric Equipment.
nuclear_acct325_misc_equip	number	FERC Account 325: Nuclear Miscellaneous Power Plant Equipment.
nuclear_acct326_asset_retirement	number	FERC Account 326: Asset Retirement Costs for Nuclear Production.
nuclear_total	number	Total Nuclear Production Plant (FERC Accounts 320-326)
other_acct340_land	number	FERC Account 340: Other Land and Land Rights.
other_acct341_structures	number	FERC Account 341: Other Structures and Improvements.
other_acct342_fuel_accessories	number	FERC Account 342: Other Fuel Holders, Products, and Accessories.
other_acct343_prime_movers	number	FERC Account 343: Other Prime Movers.
other_acct344_generators	number	FERC Account 344: Other Generators.
other_acct345_accessory_equip	number	FERC Account 345: Other Accessory Electric Equipment.
other_acct346_misc_equip	number	FERC Account 346: Other Miscellaneous Power Plant Equipment.
other_acct347_asset_retirement	number	FERC Account 347: Asset Retirement Costs for Other Production.
other_total	number	Total Other Production Plant (FERC Accounts 340-347).
production_total	number	Total Production Plant (FERC Accounts 310-347).
record_id	string	Identifier indicating original FERC Form 1 source record. format: {table
report_year	year	Four-digit year in which the data was reported.
rtmo_acct380_land	number	FERC Account 380: RTMO Land and Land Rights.
rtmo_acct381_structures	number	FERC Account 381: RTMO Structures and Improvements.
rtmo_acct382_computer_hardware	number	FERC Account 382: RTMO Computer Hardware.
rtmo_acct383_computer_software	number	FERC Account 383: RTMO Computer Software.

Field Name	Type	Description
rtmo_acct384_communication_equip	number	FERC Account 384: RTMO Communication Equipment.
rtmo_acct385_misc_equip	number	FERC Account 385: RTMO Miscellaneous Equipment.
rtmo_total	number	Total RTMO Plant (FERC Accounts 380-386)
steam_acct310_land	number	FERC Account 310: Steam Plant Land and Land Rights.
steam_acct311_structures	number	FERC Account 311: Steam Plant Structures and Improvements.
steam_acct312_boiler_equip	number	FERC Account 312: Steam Boiler Plant Equipment.
steam_acct313_engines	number	FERC Account 313: Steam Engines and Engine-Driven Generators.
steam_acct314_turbogenerators	number	FERC Account 314: Steam Turbogenerator Units.
steam_acct315_accessory_equip	number	FERC Account 315: Steam Accessory Electric Equipment.
steam_acct316_misc_equip	number	FERC Account 316: Steam Miscellaneous Power Plant Equipment.
steam_acct317_asset_retirement	number	FERC Account 317: Asset Retirement Costs for Steam Production.
steam_total	number	Total Steam Production Plant (FERC Accounts 310-317).
transmission_acct350_land	number	FERC Account 350: Transmission Land and Land Rights.
transmission_acct352_structures	number	FERC Account 352: Transmission Structures and Improvements.
transmission_acct353_station_equip	number	FERC Account 353: Transmission Station Equipment.
transmission_acct354_towers	number	FERC Account 354: Transmission Towers and Fixtures.
transmission_acct355_poles	number	FERC Account 355: Transmission Poles and Fixtures.
transmission_acct356_overhead_conductors	number	FERC Account 356: Overhead Transmission Conductors and Devices.
transmission_acct357_underground_conduit	number	FERC Account 357: Underground Transmission Conduit.
transmission_acct358_underground_conductors	number	FERC Account 358: Underground Transmission Conductors.
transmission_acct359_1_asset_retirement	number	FERC Account 359.1: Asset Retirement Costs for Transmission Plant.
transmission_acct359_roads_trails	number	FERC Account 359: Transmission Roads and Trails.
transmission_total	number	Total Transmission Plant (FERC Accounts 350-359.1)
utility_id_ferc1	integer	FERC assigned respondent_id, identifying the reporting entity. Stable fr

8.4.21 plant_unit_epa

Browse or query this table in Datasette.

Field Name	Type	Description
plant_id_epa	integer	N/A
unit_id_epa	string	Smokestack unit monitored by EPA CEMS.

8.4.22 plants_eia

Browse or query this table in Datasette.

Field Name	Type	Description
plant_id_eia	integer	The unique six-digit facility identification number, also called an ORISPL, assigned by the Energy Information Administration.
plant_id_pudl	integer	N/A
plant_name_eia	string	N/A

8.4.23 plants_eia860

Browse or query this table in Datasette.

Field Name	Type	Description
ash_impoundment	string	Is there an ash impoundment (e.g. pond, reservoir) at the plant?
ash_impoundment_lined	string	If there is an ash impoundment at the plant, is the impoundment lined?
ash_impoundment_status	string	If there is an ash impoundment at the plant, the ash impoundment status as of December 31 of the reporting year.
datum	string	N/A
energy_storage	string	Indicates if the facility has energy storage capabilities.
ferc_cogen_docket_no	string	The docket number relating to the FERC qualifying facility cogenerator status.
ferc_exempt_wholesale_generator_docket_no	string	The docket number relating to the FERC qualifying facility exempt wholesale generator status.
ferc_small_power_producer_docket_no	string	The docket number relating to the FERC qualifying facility small power producer status.
liquefied_natural_gas_storage	string	Indicates if the facility have the capability to store the natural gas in the form of liquefied natural gas.
natural_gas_local_distribution_company	string	Names of Local Distribution Company (LDC), connected to natural gas burning power plants.
natural_gas_pipeline_name_1	string	The name of the owner or operator of natural gas pipeline that connects directly to this facility or that connects to a lateral pipeline owned by this facility.
natural_gas_pipeline_name_2	string	The name of the owner or operator of natural gas pipeline that connects directly to this facility or that connects to a lateral pipeline owned by this facility.
natural_gas_pipeline_name_3	string	The name of the owner or operator of natural gas pipeline that connects directly to this facility or that connects to a lateral pipeline owned by this facility.
natural_gas_storage	string	Indicates if the facility have on-site storage of natural gas.
nerc_region	string	NERC region in which the plant is located
net_metering	string	Did this plant have a net metering agreement in effect during the reporting year? (Only displayed for facilities that report the sun or wind as an energy source). This field was only reported up until 2015
pipeline_notes	string	Additional owner or operator of natural gas pipeline.
plant_id_eia	integer	The unique six-digit facility identification number, also called an ORISPL, assigned by the Energy Information Administration.
regulatory_status_code	string	Indicates whether the plant is regulated or non-regulated.
report_date	date	Date reported.
transmission_distribution_owner_id	string	EIA-assigned code for owner of transmission/distribution system to which the plant is interconnected.
transmission_distribution_owner_name	string	Name of the owner of the transmission or distribution system to which the plant is interconnected.
transmission_distribution_owner_state	string	State location for owner of transmission/distribution system to which the plant is interconnected.
utility_id_eia	integer	EIA-assigned identification number for the company that is responsible for the day-to-day operations of the generator.
water_source	string	Name of water source associater with the plant.

8.4.24 plants_entity_eia

Browse or query this table in Datasette.

Field Name	Type	Description
balancing_authority_code_eia	string	The plant's balancing authority code.
balancing_authority_name_eia	string	The plant's balancing authority name.
city	string	The plant's city.
county	string	The plant's county.
ferc_cogen_status	string	Indicates whether the plant has FERC qualifying facility cogenerator status.
ferc_exempt_wholesale_generator	string	Indicates whether the plant has FERC qualifying facility exempt wholesale generator status
ferc_small_power_producer	string	Indicates whether the plant has FERC qualifying facility small power producer status
grid_voltage_2_kv	number	Plant's grid voltage at point of interconnection to transmission or distibution facilities
grid_voltage_3_kv	number	Plant's grid voltage at point of interconnection to transmission or distibution facilities
grid_voltage_kv	number	Plant's grid voltage at point of interconnection to transmission or distibution facilities
iso_rto_code	string	The code of the plant's ISO or RTO. NA if not reported in that year.
latitude	number	Latitude of the plant's location, in degrees.
longitude	number	Longitude of the plant's location, in degrees.
plant_id_eia	integer	The unique six-digit facility identification number, also called an ORISPL, assigned by the Energy Information Administration.
plant_name_eia	string	Plant name.
primary_purpose_naics_id	number	North American Industry Classification System (NAICS) code that best describes the primary purpose of the reporting plant
sector_id	number	Plant-level sector number, designated by the primary purpose, regulatory status and plant-level combined heat and power status
sector_name	string	Plant-level sector name, designated by the primary purpose, regulatory status and plant-level combined heat and power status
service_area	string	Service area in which plant is located; for unregulated companies, it's the electric utility with which plant is interconnected
state	string	Plant state. Two letter US state and territory abbreviations.
street_address	string	Plant street address
timezone	string	IANA timezone name
zip_code	string	Plant street address

8.4.25 plants_ferc1

Browse or query this table in Datasette.

Field Name	Type	Description
plant_id_pudl	integer	A manually assigned PUDL plant ID. May not be constant over time.
plant_name_ferc1	string	Name of the plant, as reported to FERC. This is a freeform string, not guaranteed to be consistent across references to the same plant.
utility_id_ferc1	integer	FERC assigned respondent_id, identifying the reporting entity. Stable from year to year.

8.4.26 plants_hydro_ferc1

Browse or query this table in Datasette.

Field Name	Type	Description
asset_retirement_cost	number	Cost of plant: asset retirement costs. Nominal USD.
avg_num_employees	number	Average number of employees.
capacity_mw	number	Total installed (nameplate) capacity, in megawatts.
capex_equipment	number	Cost of plant: equipment. Nominal USD.
capex_facilities	number	Cost of plant: reservoirs, dams, and waterways. Nominal USD.
capex_land	number	Cost of plant: land and land rights. Nominal USD.
capex_per_mw	number	Cost of plant per megawatt of installed (nameplate) capacity. Nominal USD.
capex_roads	number	Cost of plant: roads, railroads, and bridges. Nominal USD.
capex_structures	number	Cost of plant: structures and improvements. Nominal USD.
capex_total	number	Total cost of plant. Nominal USD.
construction_type	string	Type of plant construction ('outdoor', 'semioutdoor', or 'conventional'). Categor
construction_year	year	Four digit year of the plant's original construction.
installation_year	year	Four digit year in which the last unit was installed.
net_capacity_adverse_conditions_mw	number	Net plant capability under the least favorable operating conditions, in megawat
net_capacity_favorable_conditions_mw	number	Net plant capability under the most favorable operating conditions, in megawa
net_generation_mwh	number	Net generation, exclusive of plant use, in megawatt hours.
opex_dams	number	Production expenses: maintenance of reservoirs, dams, and waterways. Nomir
opex_electric	number	Production expenses: electric expenses. Nominal USD.
opex_engineering	number	Production expenses: maintenance, supervision, and engineering. Nominal US
opex_generation_misc	number	Production expenses: miscellaneous hydraulic power generation expenses. No
opex_hydraulic	number	Production expenses: hydraulic expenses. Nominal USD.
opex_misc_plant	number	Production expenses: maintenance of miscellaneous hydraulic plant. Nominal
opex_operations	number	Production expenses: operation, supervision, and engineering. Nominal USD.
opex_per_mwh	number	Production expenses per net megawatt hour generated. Nominal USD.
opex_plant	number	Production expenses: maintenance of electric plant. Nominal USD.
opex_rents	number	Production expenses: rent. Nominal USD.
opex_structures	number	Production expenses: maintenance of structures. Nominal USD.
opex_total	number	Total production expenses. Nominal USD.
opex_water_for_power	number	Production expenses: water for power. Nominal USD.
peak_demand_mw	number	Net peak demand on the plant (60-minute integration), in megawatts.
plant_hours_connected_while_generating	number	Hours the plant was connected to load while generating.
plant_name_ferc1	string	Name of the plant, as reported to FERC. This is a freeform string, not guarante

Table 4 – c

Field Name	Type	Description
plant_type	string	Kind of plant (Run-of-River or Storage).
project_num	integer	FERC Licensed Project Number.
record_id	string	Identifier indicating original FERC Form 1 source record. format: {table_nam
report_year	year	Four-digit year in which the data was reported.
utility_id_ferc1	integer	FERC assigned respondent_id, identifying the reporting entity. Stable from ye

8.4.27 plants_pudl

Browse or query this table in Datasette.

Field Name	Type	Description
plant_id_pudl	integer	A manually assigned PUDL plant ID. May not be constant over time.
plant_name_pudl	string	Plant name, chosen arbitrarily from the several possible plant names available in the plant matching process. Included for human readability only.

8.4.28 plants_pumped_storage_ferc1

Browse or query this table in Datasette.

Field Name	Type	Description
asset_retirement_cost	number	Cost of plant: asset retirement costs. Nominal USD.
avg_num_employees	number	Average number of employees.
capacity_mw	number	Total installed (nameplate) capacity, in megawatts.
capex_equipment_electric	number	Cost of plant: accessory electric equipment. Nominal USD.
capex_equipment_misc	number	Cost of plant: miscellaneous power plant equipment. Nominal USD.
capex_facilities	number	Cost of plant: reservoirs, dams, and waterways. Nominal USD.
capex_land	number	Cost of plant: land and land rights. Nominal USD.
capex_per_mw	number	Cost of plant per megawatt of installed (nameplate) capacity. Nominal USD.
capex_roads	number	Cost of plant: roads, railroads, and bridges. Nominal USD.
capex_structures	number	Cost of plant: structures and improvements. Nominal USD.
capex_total	number	Total cost of plant. Nominal USD.
capex_wheels_turbines_generators	number	Cost of plant: water wheels, turbines, and generators. Nominal USD.
construction_type	string	Type of plant construction ('outdoor', 'semioutdoor', or 'conventional'). Categor
construction_year	year	Four digit year of the plant's original construction.
energy_used_for_pumping_mwh	number	Energy used for pumping, in megawatt-hours.
installation_year	year	Four digit year in which the last unit was installed.
net_generation_mwh	number	Net generation, exclusive of plant use, in megawatt hours.
net_load_mwh	number	Net output for load (net generation - energy used for pumping) in megawatt-ho
opex_dams	number	Production expenses: maintenance of reservoirs, dams, and waterways. Nomin
opex_electric	number	Production expenses: electric expenses. Nominal USD.
opex_engineering	number	Production expenses: maintenance, supervision, and engineering. Nominal US
opex_generation_misc	number	Production expenses: miscellaneous pumped storage power generation expens
opex_misc_plant	number	Production expenses: maintenance of miscellaneous hydraulic plant. Nominal
opex_operations	number	Production expenses: operation, supervision, and engineering. Nominal USD.

Table 5 – c

Field Name	Type	Description
opex_per_mwh	number	Production expenses per net megawatt hour generated. Nominal USD.
opex_plant	number	Production expenses: maintenance of electric plant. Nominal USD.
opex_production_before_pumping	number	Total production expenses before pumping. Nominal USD.
opex_pumped_storage	number	Production expenses: pumped storage. Nominal USD.
opex_pumping	number	Production expenses: We are here to PUMP YOU UP! Nominal USD.
opex_rents	number	Production expenses: rent. Nominal USD.
opex_structures	number	Production expenses: maintenance of structures. Nominal USD.
opex_total	number	Total production expenses. Nominal USD.
opex_water_for_power	number	Production expenses: water for power. Nominal USD.
peak_demand_mw	number	Net peak demand on the plant (60-minute integration), in megawatts.
plant_capability_mw	number	Net plant capability in megawatts.
plant_hours_connected_while_generating	number	Hours the plant was connected to load while generating.
plant_name_ferc1	string	Name of the plant, as reported to FERC. This is a freeform string, not guarante
project_num	integer	FERC Licensed Project Number.
record_id	string	Identifier indicating original FERC Form 1 source record. format: {table_nam
report_year	year	Four-digit year in which the data was reported.
utility_id_ferc1	integer	FERC assigned respondent_id, identifying the reporting entity. Stable from ye

8.4.29 plants_small_ferc1

Browse or query this table in Datasette.

Field Name	Type	Description
capacity_mw	number	Name plate capacity in megawatts.
capex_per_mw	number	Plant costs (including asset retirement costs) per megawatt. Nominal USD.
construction_year	year	Original year of plant construction.
ferc_license_id	integer	FERC issued operating license ID for the facility, if available. This value is extracted from the original plant name where possible.
fuel_cost_per_mmbtu	number	Average fuel cost per mmBTU (if applicable). Nominal USD.
fuel_type	string	Kind of fuel. Originally reported to FERC as a freeform string. Assigned a canonical value by PUDL based on our best guess.
net_generation_mwh	number	Net generation excluding plant use, in megawatt-hours.
opex_fuel	number	Production expenses: Fuel. Nominal USD.
opex_maintenance	number	Production expenses: Maintenance. Nominal USD.
opex_total	number	Total plant operating expenses, excluding fuel. Nominal USD.
peak_demand_mw	number	Net peak demand for 60 minutes. Note: in some cases peak demand for other time periods may have been reported instead, if hourly peak demand was unavailable.
plant_name_ferc	string	PUDL assigned simplified plant name.
plant_name_original	string	Original plant name in the FERC Form 1 FoxPro database.
plant_type	string	PUDL assigned plant type. This is a best guess based on the fuel type, plant name, and other attributes.
record_id	string	Identifier indicating original FERC Form 1 source record. format: {table_name}_{report_year}_{report_prd}_{respondent_id}_{spplmnt_num}_{row_number}. Unique within FERC Form 1 DB tables which are not row-mapped.
report_year	year	Four-digit year in which the data was reported.
total_cost_of_plant	number	Total cost of plant. Nominal USD.
utility_id_ferc1	integer	FERC assigned respondent_id, identifying the reporting entity. Stable from year to year.

8.4.30 plants_steam_ferc1

Browse or query this table in Datasette.

Field Name	Type	Description
asset_retirement_cost	number	Asset retirement cost.
avg_num_employees	number	Average number of plant employees during report year.
capacity_mw	number	Total installed plant capacity in MW.
capex_equipment	number	Capital expense for equipment.

Table 6 – c

Field Name	Type	Description
capex_land	number	Capital expense for land and land rights.
capex_per_mw	number	Capital expenses per MW of installed plant capacity.
capex_structures	number	Capital expense for structures and improvements.
capex_total	number	Total capital expenses.
construction_type	string	Type of plant construction ('outdoor', 'semioutdoor', or 'conventional'). Catego
construction_year	year	Year the plant's oldest still operational unit was built.
installation_year	year	Year the plant's most recently built unit was installed.
net_generation_mwh	number	Net generation (exclusive of plant use) in MWh during report year.
not_water_limited_capacity_mw	number	Plant capacity in MW when not limited by condenser water.
opex_allowances	number	Allowances.
opex_boiler	number	Maintenance of boiler (or reactor) plant.
opex_coolants	number	Cost of coolants and water (nuclear plants only)
opex_electric	number	Electricity expenses.
opex_engineering	number	Maintenance, supervision, and engineering.
opex_fuel	number	Total cost of fuel.
opex_misc_power	number	Miscellaneous steam (or nuclear) expenses.
opex_misc_steam	number	Maintenance of miscellaneous steam (or nuclear) plant.
opex_operations	number	Production expenses: operations, supervision, and engineering.
opex_per_mwh	number	Total operating expenses per MWh of net generation.
opex_plants	number	Maintenance of electrical plant.
opex_production_total	number	Total operating epxenses.
opex_rents	number	Rents.
opex_steam	number	Steam expenses.
opex_steam_other	number	Steam from other sources.
opex_structures	number	Maintenance of structures.
opex_transfer	number	Steam transferred (Credit).
peak_demand_mw	number	Net peak demand experienced by the plant in MW in report year.
plant_capability_mw	number	Net continuous plant capability in MW
plant_hours_connected_while_generating	number	Total number hours the plant was generated and connected to load during repo
plant_id_ferc1	integer	Algorithmically assigned PUDL FERC Plant ID. WARNING: NOT STABLE
plant_name_ferc1	string	Name of the plant, as reported to FERC. This is a freeform string, not guarante
plant_type	string	Simplified plant type, categorized by PUDL based on our best guess of what w
record_id	string	Identifier indicating original FERC Form 1 source record. format: {table_nam
report_year	year	Four-digit year in which the data was reported.
utility_id_ferc1	integer	FERC assigned respondent_id, identifying the reporting entity. Stable from ye
water_limited_capacity_mw	number	Plant capacity in MW when limited by condenser water.

8.4.31 prime_movers_eia923

Browse or query this table in Datasette.

Field Name	Type	Description
abbr	string	N/A
prime_mover	string	N/A

8.4.32 purchased_power_ferc1

Browse or query this table in Datasette.

Field Name	Type	Description
billing_demand_mw	number	Monthly average billing demand (for requirements purchases, and any transactions involving demand charges). In megawatts.
coincident_peak_demand_mw	number	Average monthly coincident peak (CP) demand (for requirements purchases, and any transactions involving demand charges). Monthly CP demand is the metered demand during the hour (60-minute integration) in which the supplier's system reaches its monthly peak. In megawatts.
delivered_mwh	number	Gross megawatt-hours delivered in power exchanges and used as the basis for settlement.
demand_charges	number	Demand charges. Nominal USD.
energy_charges	number	Energy charges. Nominal USD.
non_coincident_peak_demand_mw	number	Average monthly non-coincident peak (NCP) demand (for requirements purhcases, and any transactions involving demand charges). Monthly NCP demand is the maximum metered hourly (60-minute integration) demand in a month. In megawatts.
other_charges	number	Other charges, including out-of-period adjustments. Nominal USD.
purchase_type	string	Categorization based on the original contractual terms and conditions of the service. Must be one of 'requirements', 'long_firm', 'intermediate_firm', 'short_firm', 'long_unit', 'intermediate_unit', 'electricity_exchange', 'other_service', or 'adjustment'. Requirements service is ongoing high reliability service, with load integrated into system resource planning. 'Long term' means 5+ years. 'Intermediate term' is 1-5 years. 'Short term' is less than 1 year. 'Firm' means not interruptible for economic reasons. 'unit' indicates service from a particular designated generating unit. 'exchange' is an in-kind transaction.
purchased_mwh	number	Megawatt-hours shown on bills rendered to the respondent.
received_mwh	number	Gross megawatt-hours received in power exchanges and used as the basis for settlement.
record_id	string	Identifier indicating original FERC Form 1 source record. format: {table_name}_{report_year}_{report_prd}_{respondent_id}_{spplmnt_num}_{row_number}. Unique within FERC Form 1 DB tables which are not row-mapped.
report_year	year	Four-digit year in which the data was reported.
seller_name	string	Name of the seller, or the other party in an exchange transaction.
tariff	string	FERC Rate Schedule Number or Tariff. (Note: may be incomplete if originally reported on multiple lines.)
total_settlement	number	Sum of demand, energy, and other charges. For power exchanges, the settlement amount for the net receipt of energy. If more energy was delivered than received, this amount is negative. Nominal USD.
utility_id_ferc1	integer	FERC assigned respondent_id, identifying the reporting entity. Stable from year to year.

8.4.33 transport_modes_eia923

Browse or query this table in Datasette.

Field Name	Type	Description
abbr	string	N/A
mode	string	N/A

8.4.34 utilities_eia

Browse or query this table in Datasette.

Field Name	Type	Description
utility_id_eia	integer	The EIA Utility Identification number.
utility_id_pudl	integer	A manually assigned PUDL utility ID. May not be stable over time.
utility_name_eia	string	The name of the utility.

8.4.35 utilities_eia860

Browse or query this table in Datasette.

Field Name	Type	Description
address_2	string	N/A
attention_line	string	N/A
city	string	Name of the city in which operator/owner is located
contact_firstname	string	N/A
contact_firstname_2	string	N/A
contact_lastname	string	N/A
contact_lastname_2	string	N/A
contact_title	string	N/A
contact_title_2	string	N/A
entity_type	string	Entity type of principle owner (C = Cooperative, I = Investor-Owned Utility, Q = Independent Power Producer, M = Municipally-Owned Utility, P = Political Subdivision, F = Federally-Owned Utility, S = State-Owned Utility, IND = Industrial, COM = Commercial
phone_extension_1	string	Phone extension for contact 1
phone_extension_2	string	Phone extension for contact 2
phone_number_1	string	Phone number for contact 1
phone_number_2	string	Phone number for contact 2
plants_reported_asset_manager	string	Is the reporting entity an asset manager of power plants reported on Schedule 2 of the form?
plants_reported_operator	string	Is the reporting entity an operator of power plants reported on Schedule 2 of the form?
plants_reported_other_relationship	string	Does the reporting entity have any other relationship to the power plants reported on Schedule 2 of the form?
plants_reported_owner	string	Is the reporting entity an owner of power plants reported on Schedule 2 of the form?
report_date	date	Date reported.
state	string	State of the operator/owner
street_address	string	Street address of the operator/owner
utility_id_eia	integer	EIA-assigned identification number for the company that is responsible for the day-to-day operations of the generator.
zip_code	string	Zip code of the operator/owner
zip_code_4	string	N/A

8.4.36 utilities_entity_eia

Browse or query this table in Datasette.

Field Name	Type	Description
utility_id_eia	integer	The EIA Utility Identification number.
utility_name_eia	string	The name of the utility.

8.4.37 utilities_ferc1

Browse or query this table in Datasette.

Field Name	Type	Description
utility_id_ferc1	integer	FERC assigned respondent_id, identifying the reporting entity. Stable from year to year.
utility_id_pudl	integer	A manually assigned PUDL utility ID. May not be stable over time.
utility_name_ferc1	string	Name of the responding utility, as it is reported in FERC Form 1. For human readability only.

8.4.38 utilities_pudl

Browse or query this table in Datasette.

Field Name	Type	Description
utility_id_pudl	integer	A manually assigned PUDL utility ID. May not be stable over time.
utility_name_pudl	string	Utility name, chosen arbitrarily from the several possible utility names available in the utility matching process. Included for human readability only.

8.4.39 utility_plant_assn

Browse or query this table in Datasette.

Field Name	Type	Description
plant_id_pudl	integer	N/A
utility_id_pudl	integer	N/A

8.5 Contributing to PUDL

Welcome! We're excited that you're interested in contributing to the Public Utility Data Liberation effort! The work is currently being coordinated by the members of the Catalyst Cooperative. PUDL is meant to serve a wide variety of public interests including academic research, climate advocacy, data journalism, and public policy making. This open source project has been supported by a combination of volunteer contributions, grant funding from the Alfred P. Sloan Foundation, and reinvestment of net income from the cooperative's client projects.

Please make sure you review our *code of conduct*, which is based on the Contributor Covenant. We want to make the PUDL project welcoming to contributors with different levels of experience and diverse personal backgrounds.

8.5.1 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 create new example notebooks
- Help us create more and better software *test cases*.
- 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 related to open energy data, open source energy system modeling, how energy policy can be affected by better data, or open source tools we should check out.
- Cite PUDL using DOIs from Zenodo if you use the software or data in your own published work.
- Point us toward appropriate grant funding opportunities and meetings where we might present our work.
- Share your Jupyter notebooks and other analyses that use PUDL.
- Hire Catalyst to do analysis for your organization using the PUDL data – contract work helps us self-fund ongoing open source development.
- Contribute code via pull requests. See the *developer setup* for more details.
- And of course. . . we also appreciate financial contributions.

See also:

- *Development Setup* for instructions on how to set up the PUDL development environment.

8.5.2 Find us on 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.

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 data work from being activists and policy wonks – so it's easy for us 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!

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, 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.

8.6 Development

8.6.1 Development Setup

This page will walk you through what you need to do if you want to be able to contribute code or documentation to the PUDL project.

These instructions assume that you are working on a Unix-like operating system (MacOS or Linux) and are already familiar with `git`, GitHub, and the Unix shell.

> **Warning:** While it should be possible to set up the development environment on Windows, we haven't done it. In the future we may create a Docker image that provides the development environment. E.g. for use with VS Code's Containers extension.

Note: If you're new to `git` and GitHub , you'll want to check out:

- The Github Workflow
- Collaborative Development Models
- Forking a Repository
- Cloning a Repository

Install conda

We use the `conda` package manager to specify and update our development environment, preferentially installing packages from the community maintained conda-forge distribution channel. We recommend using miniconda rather than the large pre-defined collection of scientific packages bundled together in the Anaconda Python distribution. You may also want to consider using mamba – a faster drop-in replacement for `conda` written in C++.

After a conda package manager, make sure it's configured to use strict channel priority with the following commands:

```
$ conda update conda
$ conda config --set channel_priority strict
```

Fork and Clone the PUDL Repository

Unless you're part of the Catalyst Cooperative organization already, you'll need to fork the PUDL repository This makes a copy of it in your personal (or organizational) account on GitHub that is independent of, but linked to, the original "upstream" project.

Then, clone the repository from your fork to your local computer where you'll be editing the code or docs. This will download the whole history of the project, including the most recent version, and put it in a local directory where you can make changes.

Create the PUDL Dev Environment

Inside the `devtools` directory of your newly cloned repository, you'll see an `environment.yml` file, which specifies the `pudl-dev conda` environment. You can create and activate that environment from within the main repository directory by running:

```
$ conda update conda
$ conda env create --name pudl-dev --file devtools/environment.yml
$ conda activate pudl-dev
```

This environment installs the `catalystcoop.pudl` package directly using the code in your cloned repository so that it can be edited during development. It also installs all of the software PUDL depends on, some packages for testing and quality control, working with interactive Jupyter Notebooks, and a few Python packages that have binary dependencies which can be easier to satisfy through `conda` packages.

Updating the PUDL Dev Environment

Periodically you will need to update your development (`pudl-dev`) conda environment. This will get you newer versions of existing dependencies, and also incorporate any changes to the environment specification that have been made by other contributors. The most reliable way to do this is to remove the existing environment and recreate it.

Note: Different development branches within the repository may specify their own slightly different versions of the `pudl-dev` conda environment. As a result you may need to update your environment when switching from one branch to another.

If you want to work with the most recent version of the code on a branch named `new-feature`, then from within the top directory of the PUDL repository you would do:

```
$ git checkout new-feature
$ git pull
$ conda deactivate
$ conda update conda
$ conda env remove --name pudl-dev
$ conda env create --name pudl-dev --file devtools/environment.yml
$ conda activate pudl-dev
```

If you find yourself recreating the environment frequently, and are frustrated by how long it takes `conda` to solve the dependencies, we recommend using the mamba solver. You'll want to install it in your `base` conda environment – i.e. with no conda environment activated):

```
$ conda deactivate
$ conda install mamba
```

Then the above development environment update process would become:

```
$ git checkout new-feature
$ git pull
$ conda deactivate
$ mamba update mamba
$ mamba env remove --name pudl-dev
$ mamba env create --name pudl-dev --file devtools/environment.yml
$ conda activate pudl-dev
```

If you are working with locally processed data and there have been changes to the expectations about that data in the PUDL software, you may also need to regenerate your PUDL SQLite database or other outputs. See *Running the ETL Pipeline* for more details.

Set Up Code Linting

We use several automated tools to apply uniform coding style and formatting across the project codebase. This is known as code linting and it reduces merge conflicts, makes the code easier to read, and helps catch some types of bugs before they are committed. These tools are part of the `pudl-dev` 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.

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 before publishing them.

To make sure they are run before you commit any code, you need to enable the pre-commit hooks scripts with this command:

```
$ pre-commit install
```

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

See also:

- The pre-commit project: A framework for managing and maintaining multi-language pre-commit hooks.
- Real Python Code Quality Tools and Best Practices gives a good overview of available linters and static code analysis tools.

Code and Docs Linters

Flake8 is a popular Python linting framework, with a large selection of plugins. We use it to check the formatting and syntax of the code and docstrings embedded within the PUDL packages. 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.

Automatic Formatting

Rather than alerting you that there's a style issue in your Python code, autopep8 tries to fix it for you automatically, applying consistent formatting rules based on **PEP 8**. Similarly isort automatically groups and orders Python import statements in each module to minimize diffs and merge conflicts.

Linting Within Your Editor

If you are using an editor designed for Python development many of these code linting and formatting tools can be run automatically in the background while you write code or documentation. Popular editors that work with the above tools include:

- Visual Studio Code, from Microsoft (free)
- Atom developed by GitHub (free), and
- Sublime Text (paid).

Each of these editors have their own collection of plugins and settings for working with linters and other code analysis tools.

See also:

Real Python Guide to Code Editors and IDEs

Creating a Workspace

PUDL needs to know where to store its big piles of inputs and outputs. It also comes with some example configuration files. 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 where any outputs that are generated end up. The script will also put a configuration file in your home directory, called `.pudl.yml` which 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 working directory.

The workspace is laid out like this:

Directory / File	Contents
`data/`	Raw data, automatically organized by source, year, etc.
`datapkg/`	Tabular data packages generated by PUDL.
`parquet/`	Apache Parquet files generated by PUDL.
`settings/`	Example configuration files for controlling PUDL scripts.
`sqlite/`	`sqlite3` databases generated by PUDL.

8.6.2 Settings Files

Several of the scripts provided as part of PUDL require more arguments than can be easily managed on the command line. It's also useful to preserve a record of how the data processing pipeline was run in one instance so it can be re-run in exactly the same way. We have these scripts read their settings from YAML files, examples of which are included in the distribution.

There are two example files that are deployed into a users workspace with the `pudl_setup` script (see: *Creating a Workspace*). The two settings files direct PUDL to process 1 year ("fast") and all years ("full") of data respectively. Each file contains parameters for both the `ferc1_to_sqlite` and the `pudl_etl` scripts.

Setttings for ferc1_to_sqlite

Parameter	Description
`ferc1_to_sqlite_refyear`	A single 4-digit year to use as the reference for inferring FERC Form 1 database's structure. Typically the most recent year of available data.
`ferc1_to_sqlite_years`	A list of years to be included in the cloned FERC Form 1 database. You should only use a continuous range of years. 1994 is the earliest year available.
`ferc1_to_sqlite_tables`	A list of strings indicating what tables to load. The list of acceptable tables can be found in the the example settings file and corresponds to the values found in the `ferc1_dbf2tbl` dictionary in *pudl.constants*.

Settings for pudl_etl

The `pudl_etl` script requires a YAML settings file. In the repository this example file is lives in `src/pudl/package_data/settings`. This example file (`etl_example.yml`) is deployed onto a user's system in the `settings` directory within the PUDL workspace when the `pudl_setup` script is run. Once this file is in the settings directory, users can copy it and modify it as appropriate for their own use.

This settings file allows users to determine the scope of the integrated by PUDL. Most datasets can be used to generate stand-alone data packages. If you only want to use FERC Form 1, you can remove the other data package specifications, or alter their parameters such that none of their data is processed (e.g. by setting the list of years to be an empty list). 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 because it relies on IDs that come from EIA 860. Similarly, EIA Forms 860 and 923 are very tightly related. You can load only EIA 860, but the settings verification will automatically add in a few 923 tables that are needed to generate the complete list of plants and generators.

Warning: If you are processing the EIA 860/923 data, we **strongly recommend** including the same years in both datasets. We only test two combinations of inputs:

- That **all** available years of EIA 860/923 can be processed together, and
- That the most recent year of both datasets can be processed together.

Other combinations of years may yield unexpected results.

Structure of the pudl_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 can be edited. There are two high-level elements of the settings file which pertain to the entire bundle of tabular data packages which will be generated: `datapkg_bundle_name` and `datapkg_bundle_settings`. The `datapkg_bundle_name` determines which directory the data packages are written into. The elements and structure of the `datapkg_bundle_settings` are described below:

```
datapkg_bundle_settings
  ├── name : unique name identifying the data package
  │   title : short human readable title for the data package
  │   description : a longer description of the data package
  │   datasets
```

(continues on next page)

(continued from previous page)

```
│       ├── dataset name
│       │   ├── dataset etl parameter (e.g. states) : list of states
│       │   └── dataset etl parameter (e.g. years) : list of years
│       └── dataset name
│       │   ├── dataset etl parameter (e.g. states) : list of states
│       │   └── dataset etl parameter (e.g. years) : list of years
└── another data package...
```

The dataset names must not be changed. The dataset names enabled include: `eia` (which includes Forms 860/923 only for now), `ferc1`, and `epacems`. Any other dataset name will result in an assertion error.

Note: We strongly recommend leaving the arguments that specify which database tables are generated unchanged – i.e. always include all of the tables, as many analyses require data from multiple tables, and removing a few tables doesn't change how long the ETL process takes by much.

Dataset ETL parameters (like years, states, tables), will only register if they are a part of the correct dataset. If you put some FERC Form 1 ETL parameter in an EIA dataset specification, FERC Form 1 will not be loaded as a part of that dataset. For an exhaustive listing of the available parameters, see the `etl_example.yml` file.

8.6.3 Running the ETL Pipeline

So you want to run the PUDL data processing pipeline? This is the most involved way to get access to PUDL data. It's only recommended if you want to edit the ETL process or contribute to the code base. Check out the *Data Access* documentation if you just want to use the processed data.

These instructions assume you have already gone through the development setup (see: *Development Setup*).

There are four main scripts that are involved in the PUDL processing pipeline:

1. `ferc1_to_sqlite` *converts the FERC Form 1 DBF files* into a single large SQLite database so that the data is easier to extract.
2. `pudl_etl` is where the magic happens. This is the main script which coordinates the "Extract, Transform, Load" process that generates Tabular Data Packages.
3. `datapkg_to_sqlite` converts the Tabular Data Packages into a SQLite database. We recommend doing this for all of the smaller to medium sized tables, which is currently everything but the hourly EPA CEMS data.
4. `epacems_to_parquet` converts the (~1 billion row) EPA CEMS Data Package into Apache Parquet files for fast on-disk querying.

Settings files dictate which datasets, years, tables, or states get run through the the processing pipeline. Two example settings files are provided in the `settings` folder that is created when you run `pudl_setup`.

See also:

- *Creating a Workspace* for more on how to create a PUDL data workspace.
- *Settings Files* for info details on the contents of the settings files.

The Fast ETL

Running the fast ETL processes one year of data for each dataset. This is what we do in our *software integration tests*.

```
$ ferc1_to_sqlite settings/etl_fast.yml
$ pudl_etl settings/etl_fast.yml
$ datapkg_to_sqlite \
    datapkg/pudl-fast/ferc1/datapackage.json \
    datapkg/pudl-fast/epacems-eia/datapackage.json
$ epacems_to_parquet datapkg/pudl-fast/epacems-eia/datapackage.json
```

The Full ETL

The full ETL setting file includes all the datasets with all of the years and tables with the exception of EPA CEMS. A full ETL for EPA CEMS can take up to 15 hours of processing time so the example setting here is all years of CEMS for one state (Idaho!) which takes around 20 minutes to process.

```
$ ferc1_to_sqlite settings/etl_full.yml
$ pudl_etl settings/etl_full.yml
$ datapkg_to_sqlite datapkg/pudl-full/ferc1/datapackage.json \
    datapkg/pudl-full/eia/datapackage.json
$ epacems_to_parquet datapkg/pudl-full/epacems-eia/datapackage.json
```

Additional Notes

These commands should result in a bunch of Python `logging` output describing what the script is doing, and file outputs in the `sqlite`, `datapkg`, and `parquet` directories within your workspace. When the ETL is complete you should see new files at `sqlite/ferc1.sqlite` and `sqlite/pudl.sqlite`, and a new directory at `datapkg/pudl-fast` or `datapkg/pudl-full` containing several datapackage directories – one for each of the `ferc1`, `eia` (Forms 860 and 923), and `epacems-eia` datasets.

Each of the data packages that are part of the bundle have metadata describing their structure. This metadata is stored in the associated `datapackage.json` file. The data are 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 either of the settings files and running the script again with your new settings file as an argument. Comments in the example settings file explain the available parameters. Know that these example files are the only configurations that are tested automatically and supported.

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 of the output datapackage bundle in the configuration file.

All of the PUDL scripts have help messages if you want additional information (run `script_name --help`).

8.6.4 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, as well as Zenhub which provides additional features for sprint planning, task estimation, and progress reports.

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. Try to make sure that issues have informative tags so we can find them easily.

We use Zenhub Sprints, Epics, and Releases to track our progress. These won't be visible unless you have the ZenHub browser extension installed.

GitHub Workflow

- We have 2 persistent branches: **main** and **dev**.
- We create temporary feature branches off of **dev** and make pull requests to **dev** throughout our 2 week long sprints.
- At the end of each sprint, assuming all the tests are passing, **dev** is merged into **main**.

Pull Requests

- Before making a PR, make sure the tests run and pass locally, including the code linters and pre-commit hooks. See *Set Up Code Linting* for details.
- Don't forget to merge any new commits to the **dev** branch into your feature branch before making a PR.
- If for some reason the continuous integration tests fail for your PR, try and figure out why and fix it, or ask for help. If the tests fail we don't want to merge it into **dev**. You can see the status of the CI builds in the GitHub Actions for the PUDL repo.
- Please don't decrease the overall test coverage – if you introduce new code it also needs to be exercised by the tests. See *Testing PUDL* for details.
- Write good docstrings, using the Google format
- Pull Requests should update the documentation to reflect changes to the code, especially if it changes something user facing, like how one of the command line scripts works.

Releases

- Periodically, we tag a new release on **main** and upload the packages to the Python Package Index and conda-forge.
- Whenever we tag a release on Github, the repository is archived on Zenodo and issued a DOI.
- For some software releases, we archive processed data on Zenodo along with a Docker container that encapsulates the necessary software environment.

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.

8.6.5 Testing PUDL

We use Tox to coordinate our software testing, and to manage other build and sanity checking tools. Under the hood it invokes a variety of other collections of command-line tools in predefined combinations that are described in `tox.ini`. These include software tests defined using pytest, code linters like `flake8`, documentation generators like Sphinx, and sanity checks defined as git pre-commit hooks. Each of these tools, or sometimes collections of related tools, can be selected at the command line. They can also be run independently without using Tox, but for the sake of simplicitly and standardization, we try to mostly just run them using the predefined settings we have configured in Tox.

The simplest way to test PUDL – which is also how the code is tested automatically by our continuous integration setup – is to just run Tox alone with no arguments. This will typically take 25 minutes to run.

```
$ tox
```

Note: If you aren't familiar with pytest and Tox already, you may want to go peruse their introductory documentation.

- Getting Started with pytest
- Tox Documentation

Software Tests

Our `pytest` based software tests are all stored under the `test/` directory in the main repository. They are organized into 3 broad categories, each with its own subdirectory:

- **Software Unit Tests** (`test/unit/`) can be run in seconds and don't require any external data. They test the basic functionality of various functions and classes, often using minimal inline data structures that are specified in the test modules themselves.
- **Software Integration Tests** (`test/integration/`) test larger collections of functionality, including the interactions between different parts of the overall software system, and in some cases interactions with external systems, requiring network connectivity. The main thing our integration tests do is run the full PUDL data processing pipeline for the most recent year of data. This takes around 15 minutes.
- **Data Validations** (`test/validate/`) sanity check the PUDL outputs generated by the data processing pipeline. This helps us catch issues with the input data, and more subtle bugs that don't prevent the code from executing, but do have unintended or unexpected impacts on the output data. The data validation requires a fully populated PUDL database and is quite different from the other tests.

Running tests with Tox

Tox installs the PUDL package in a fresh Python environment, ensuring that the tests only have access to packages which would be installed on a new user's computer. Tox's overall behavior is configured with the `tox.ini` file in the main repository directory. There are several different "test environments" defined, to test different aspects of the software, or to perform other actions like building the documentation. We'll go through some of the most common ones below.

Continuous Integration Tests

Our default tox test environment is `ci` – which includes all of the tests that will be run in continuous integration using a GitHub Action. You should run these tests before pushing code to the repository or making a pull request. Because it's the default test environment, it will be run if you call Tox without any arguments:

```
$ tox
```

This is equivalent to:

```
$ tox -e ci
```

If the PUDL package's dependencies have been changed (in `setup.py`), or you recently ran the tests while on another branch of the repository with other dependencies, you may need to tell Tox to recreate the software environment it uses with the `-r` flag. This behavior is turned on by default for the `ci`, `full`, and `validate` tests, since they take a long time to run and the extra time required to recreate the software environment is short by comparison.

In addition to running the `unit` and `integration` tests, the CI test environment lints the code and documentation input files, and uses Sphinx to build the documentation. It also generates a test coverage report. Running the full set of CI tests takes 20-25 minutes, and requires a fair amount of data. If you don't already have that data downloaded, it will be downloaded automatically and put in your *local datastore*

Note: Locally the tests will run using whatever version of Python is part of your `pudl-dev` conda environment, but we have our CI set up to test on both Python 3.8 and 3.9 in parallel.

Software Unit and Integration Tests

To run the `unit` or `integration` tests on their own, you use the `-e` flag to choose those test environments explicitly:

```
$ tox -e unit
```

or:

```
$ tox -e integration
```

Full ETL Tests

As mentioned above, the CI tests process a single year of data. If you would like to more exhaustively test the ETL process without affecting your existing FERC 1 and PUDL databases, you can use the `full` test environment, which may take close to an hour to run:

```
$ tox -e full
```

This will process *all years of data* for the EIA and FERC datasets, and all years of EPA CEMS data for a single state (Idaho). The ETL parameters for this test are defined in `test/settings/full-integration-tests.yml`

Running Other Commands with Tox

You can run any of the individual test environments that `tox -av` lists on their own:

```
$ tox -av

default environments:
ci                -> Run all continuous integration (CI) checks & generate test␣
↪coverage.

additional environments:
flake8            -> Run the full suite of flake8 linters on the PUDL codebase.
pre_commit        -> Run git pre-commit hooks not covered by the other linters.
bandit            -> Check the PUDL codebase for common insecure code patterns.
linters           -> Run the pre-commit, flake8 and bandit linters.
doc8              -> Check the documentation input files for syntactical correctness.
docs              -> Remove old docs output and rebuild HTML from scratch with Sphinx
unit              -> Run all the software unit tests.
ferc1_solo        -> Test whether FERC 1 can be loaded into the PUDL database alone.
integration       -> Run all software integration tests and process a full year of␣
↪data.
validate          -> Run all data validation tests. This requires a complete PUDL DB.
ferc1_schema      -> Verify FERC Form 1 DB schema are compatible for all years.
full_integration -> Run ETL and integration tests for all years and data sources.
full              -> Run all CI checks, but for all years of data.
build             -> Prepare Python source and binary packages for release.
testrelease       -> Do a dry run of Python package release using the PyPI test server.
release           -> Release the PUDL package to the production PyPI server.
```

Note that not all of them literally run tests. For instance, to lint and build the documentation you can run:

```
$ tox -e docs
```

To run all of the code and documentation linters, but not run any of the other tests:

```
$ tox -e linters
```

Each of the test environments defined in `tox.ini` is just a collection of dependencies and commands. To see what they consist of, you can open the file in your text editor. Each section starts with `[testenv:xxxxxx]` and the section called `commands` is a list of shell commands that that test environment will run.

Selecting Input Data for Integration Tests

The software integration tests need a year's worth of input data to process. By default they will look in your local PUDL datastore to find it. If the data they need isn't available locally, they will download it from Zenodo and put it in the local datastore.

However, if you're editing code that affects how the datastore works, you probably don't want to risk contaminating your working datastore. You can use a disposable temporary datastore instead by having Tox pass the `--tmp-data` flag in to `pytest` like this:

```
$ tox -e integration -- --tmp-data
```

The floating `--` isn't a typo, it tells Tox that you're done giving it command line arguments, and that any additional arguments it gets should be passed through to `pytest`. We've configured `pytest` (through the `test/conftest.py` configuration file) to be on the lookout for the `--tmp-data` flag and act accordingly.

See also:

- *Development Setup* for more on how to set up a PUDL workspace, including a datastore.
- *Working with the Datastore* for more on how to work with the datastore.

Data Validation

Given the processed outputs of the PUDL ETL pipeline, we have a collection of tests that can be run to verify that the outputs look correct. We run all available data validations before each data release is archived on Zenodo. It is useful to run the data validation tests prior to making a pull request that makes changes to the ETL process or output functions, to ensure that the outputs have not been unintentually affected.

These data validation tests are organized into datasource specific modules under `test/validate`. Running the full data validation can take as much as an hour, depending on your computer. These tests require a fully populated PUDL database which contains all available FERC and EIA data, as specified by the `test/settings/full-integration-test.yml` input file. They are run against the "live" SQLite database in your pudl workspace at `sqlite/pudl.sqlite`. To run the full data validation against an existing database:

```
$ tox -e validate
```

The data validation cases that pertain to the contents of the data tables are currently stored as part of the *`pudl.validate`* module.

The expected number of records in each output table is stored in the validation test modules under `test/validate` as pytest parameterizations.

Data Validation Notebooks

We have a collection of Jupyter Notebooks that run the same functions as the data validation. The notebooks 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`

Like the data validations, the notebooks will only run successfully when there's a full PUDL SQLite database available in your PUDL workspace.

Running pytest Directly

Running tests directly with `pytest` gives you the ability to run only tests from a particular test module, or even a single individual test case. It's also faster because there's no testing environment to set up. Instead, it just uses your Python environment, which should be the `pudl-dev` conda environment discussed in *Development Setup*. This is convenient if you're debugging something specific, or developing new test cases, but it's not as robust as using Tox.

Running specific tests

To run the software unit tests with `pytest` directly (the same set of tests that would be run by `tox -e unit`):

```
$ pytest test/unit
```

To run only the unit tests for the Excel spreadsheet extraction module:

```
$ pytest test/unit/extract/excel_test.py
```

To run only the unit tests defined by a single test class within that module:

```
$ pytest test/unit/extract/excel_test.py::TestGenericExtractor
```

Custom PUDL pytest flags

We have defined several custom flags to control pytest's behavior when running the PUDL tests. They are mostly intended for use internally, to specify the behavior we want in the high level Tox test environments.

You can always check to see what custom flags exist by running `pytest --help` and looking at the `custom options` section:

```
custom options:
--live-dbs            Use existing PUDL/FERC1 DBs instead of creating temporary ones.
--tmp-data            Download fresh input data for use with this test run only.
--etl-settings=ETL_SETTINGS
                      Path to a non-standard ETL settings file to use.
--gcs-cache-path=GCS_CACHE_PATH
                      If set, use this GCS path as a datastore cache layer.
--sandbox             Use raw inputs from the Zenodo sandbox server.
```

The main flexibility that these custom options provide is in selecting where the raw input data comes from, and what data the tests should be run against. Being able to specify the tests to run and the data to run them against independently simplifies the test suite, and keeps the data and tests very clearly separated.

The `--live-dbs` option lets you use your existing FERC 1 and PUDL databases instead of building a new database at all. This can be useful if you want to test code that only operates on an existing database, and has nothing to do with the construction of that database. For example, the output routines:

```
$ pytest --live-dbs test/integration/fast_output_test.py
```

We also use this option to run the data validations.

Assuming you do want to run the ETL and build new databases as part of the test you're running, the contents of that database are determined by an ETL settings file. By default, the settings file that's used is `test/settings/integration-test.yml` But it's also possible to use a different input file, generating a different database, and then run some tests against that database.

For example, we test that FERC 1 data can be loaded into a PUDL database all by itself by running the ETL tests with a settings file that includes only A couple of FERC 1 tables for a single year. This is the `ferc1_solo` Tox test environment:

```
$ pytest --etl-settings=test/settings/ferc1-solo-test.yml test/integration/etl_test.py
```

Similarly, we use the `test/settings/full-integration-test.yml` settings file to specify an exhaustive collection of input data, and then run a test that checks that the database schemas extracted from all historical FERC 1 databases are compatible with each other. This is the `ferc1_schema` test:

```
$ pytest --etl-settings test/settings/full-integration-test.yml test/integration/etl_
↪test.py::test_ferc1_schema
```

The raw input data that all the tests use is ultimately coming from our archives on Zenodo. but you can optionally tell the tests to look in a different places for more rapidly accessbile caches of that data, and to force the download of a fresh copy (especially useful when you are testing the datastore functionality specifically). By default the tests will use the datastore that's part of your local PUDL workspace.

For example, to run the ETL portion of the integration tests, and download fresh input data to a temporary datastore that's later deleted automatically:

```
$ pytest --tmp-data test/integration/etl_test.py
```

8.6.6 Building the Documentation

We use Sphinx and Read The Docs to semi-automatically build and host our documentation.

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`.

If you are editing the documentation, and need to regenerate the outputs as you go to see your changes reflected locally, the most reliable option is to use Tox, which will remove the previously generated outputs, and regenerate everything from scratch:

```
$ tox -e docs
```

If you're just working on a single page and don't care about the entire set of documents being regenerated and linked together, you can call Sphinx directly:

```
$ 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.

To view the documentation that's been output at HTML you'll need to open the `docs/_build/html/index.html` file within the PUDL repository with a web browser. You may also be able to set up automatic previewing of the rendered documentation in your text editor with appropriate plugins.

Note: Some of the documentation files are dynamically generated. We use the sphinx-apidoc utility to generate RST files from the docstrings embedded in our source code, so you should never edit the files under `docs/api`. If you create a new module, the corresponding documentation file will also need to be checked in to version control.

Similarly the *PUDL Data Dictionary* is generated dynamically by the `pudl.convert.datapkg_to_rst` script, which is run by Tox when it builds the docs.

8.6.7 Working with the Datastore

The input data that PUDL processes comes from a variety of US government agencies. However, these agencies typically make the data available on their websites or via FTP without planning for programmatic access. To ensure reproducible, programmatic access, we periodically archive the input files on the Zenodo research archiving service maintained by CERN. (See our pudl-scrapers and pudl-zenodo-storage repositories on GitHub for more information.)

When PUDL needs a data resource it will attempt to automatically retrieve it from Zenodo and store it locally in a file hierarchy organized by dataset and the versioned DOI of the corresponding Zenodo deposition.

The `pudl_datastore` script can also be used to pre-download the raw input data in bulk. 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 PUDL *Data Sources*. At present the `pudl_datastore` script downloads the entire collection of data available for each dataset. For the FERC Form 1 and EPA CEMS datasets, this is several gigabytes.

For example, to download the full *EIA Form 860* dataset (covering 2001-present) you would use:

```
$ pudl_datastore --dataset eia860
```

For more detailed usage information, see:

```
$ pudl_datastore --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/censusdp1tract/
data/eia860/
data/eia861/
data/eia923/
data/epacems/
data/ferc1/
data/ferc714/
```

If the download fails to complete successfully, the script can be run repeatedly until all the files are downloaded. It will not try and re-download data which is already present locally.

Adding a new Dataset to the Datastore

There are three components necessary to prepare a new datastet for use with the PUDL datastore.

1. Create a `pudl-scraper` to download the raw data.
2. Use `pudl-zenodo-storage` to upload the data to Zenodo.
3. Prepare the datastore to retrieve the data from Zenodo.

In the event that data is already available on Zenodo in the appropriate format, it may be possible to skip steps 1 and 2.

Create a scraper

Where possible, we use Scrapy to handle data collection. Our scrapy spiders, as well as any custom scripts, are located in our scrapers repo. Familiarize yourself with scrapy, and note the following.

From a scraper, a correct ouput directory takes the form:

```
`pudl_scrapers.helpers.new_output_dir(self.settings["OUTPUT_DIR"] /
"datastet_name")`
```

The `pudl_scrapers.settings` and `pudl_scrapers.helpers` can be imported outside the context of a Scrapy scraper to achieve the same effect as needed.

To take advantage of the existing file saving pipeline, create a custom item in the `items.py` collection. Make sure that it inherits from the existing `DataFile` class, and ensure that your spider yields the new item. See the `items.py` for examples.

If you follow those guidelines your new scraper should play well with the rest of the environment.

Prepare zenodo_store

Our zenodo_store script initializes and updates data sources that we maintain on Zenodo . It prepares Frictionless Datapackages from scraped files and uploads them to the appropriate Zenodo archive.

To add a new archive to our Zenodo storage collection:

- **Update `zs.metadata` with a UUID and metadata for the new Zenodo archive.** These details will be used by Zenodo to identify and describe the archive on the website. The UUID is used to uniquely distinguish the archive **prior to the creation of a DOI.**
- Prepare a new library to handle the **frictionless datapackage** descriptor of the archive.
  - The library name should take the form `frictionless.DATASET_raw`.
  - The library must contain frictionless data metadata describing the archive.
  - The library must contain a `datapackager(dfiles)` function that:
    * receives a list of zenodo file descriptors
    * converts each to an appropriate frictionless datapackage resource descriptor
    * **Important**: The resource descriptor must include an additional `descriptor["remote_url"]` that contains the zenodo url to download its resource. This will be the same as the `descriptor["path"]` at this stage.
    * If there are criteria by which you wish to be able to discover or filter specific resources, `descriptor["parts"][...]` should be used to denote those details. For example, `descriptor["parts"]["year"] = 2018` would be appropriate to allow filtering by year.
    * Combines the resource descriptors and frictionless metadata to produce the complete datapackage descriptor as a python dict.
- In the `bin/zenodo_store.py` script:
  - Import the new frictionless library.
  - Add the new source to the `archive_selection` function; follow the format of the existing selectors.
  - Add the new source name to the help text in the `parse_main() .. deposition` argument.

The above steps should be sufficient to allow automatic initialization and updates of the new data source on Zenodo.

You initialize an archive (preferably starting with the sandbox) by running `zenodo_store.py --initialize --verbose --sandbox`

If successful, the DOI and url for your archive will be printed. You will need to visit the url to review and publish the Zenodo archive before it can be used.

If you lose track of the DOI, you can look up the archive on Zenodo using the UUID from `zs.metadata`.

Prepare the Datastore

If you have used a scraper and zenodo_store to prepare a Zenodo archive as above, you can add support for your archive to the datastore by adding the DOI to pudl.workspace.datastore.DOI, under "sandbox" or "production" as appropriate.

If you want to prepare an archive for the datastore separately, the following are required.

#. The root path must contain a `datapackage.json` file that conforms to the frictionless datapackage spec #. Each listed resource among the `datapackage.json` resources must include:

- `path` containing the zenodo download url for the specific file.
- `remote_url` with the same url as the `path`
- `name` of the file
- `hash` with the md5 hash of the file
- `parts` a set of key / value pairs defining additional attributes that can be used to select a subset of the whole datapackage. For example, the `epacems` dataset is partitioned by year and state, and `"parts": {"year": 2010, "state": "ca"}` would indicate that the resource contains data for the state of California in the year 2010. Unpartitioned datasets like the `ferc714` which includes all years in a single file, would have an empty `"parts": {}`

8.6.8 Cloning the FERC Form 1 DB

FERC Form 1 is. . . special.

The *FERC Form 1* 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/etl-full.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_stmnt`) 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.

Note: There are a handful of `respondent_id` values which appear in the FERC Form 1 database tables, but which do not show up in `f1_respondent_id`. This renders the foreign key relationships between those tables invalid. During the database cloning process we add these `respondent_id` values to the `f1_respondent_id` table, with a `respondent_name` indicating that the ID was filled in by PUDL.

8.6.9 Naming Conventions

In the PUDL codebase, we aspire to follow the naming and other conventions detailed in **PEP 8**.

Admittedly we have a lot of… named things in here, and we haven't been perfect about following conventions everywhere. We're trying to clean things up as we come across them again in maintaining the code.

- 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 (i.e. `snake_case`).
- Semi-private helper functions (functions used within a single module only and not exposed via the public API) 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`).

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
`equip`	equipment
`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
`frc_eia923`	Fuel Receipts and Costs (*EIA Form 923*)
`gen_eia923`	Generation (*EIA Form 923*)
`gf_eia923`	Generation Fuel (*EIA Form 923*)
`gens_eia923`	Generators (*EIA Form 923*)
`utils_eia860`	Utilities (*EIA Form 860*)
`own_eia860`	Ownership (*EIA Form 860*)

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.

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`).

Table Names

See this article on database naming conventions.

- Table names in snake_case
- The data source should follow the thing it applies to e.g. `plant_id_ferc1`

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 nominal 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.

8.6.10 Data and ETL Design Guidelines

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

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

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. For the sake of reproducibility, it's important that we archive the inputs alongside the ouputs – since the reporting agencies often go back and update the data they have published without warning, and without version control.

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 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.

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 pace – 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.

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).

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 *Farenheit*?! What century even is this?).

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.

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.

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 Hourly*, *FERC EQR*). 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.

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`, `eia861`, etc. See the *naming conventions* document for more details.

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.

8.6.11 Packaging and Dependencies

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 define its dependencies. When we first packaged up PUDL Python packaging systems were a bit of a mess. Changes to the Python packaging & build system implemented as a result of **PEP 517** and **PEP 518** have improved the available options and we should look at using a simpler more modern setup. The online Python Packages book is a great guide to current best / better practices.

`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 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='>=3.8'`: Specifies what versions of Python the package is expected to run on. In this case, it's anything greater than or equal to 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 admittedly convoluted.
- `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. For now this is mostly used in conjunction with Tox, to ensure that the required documentation and testing packages are installed alongside PUDL in the virtual environment.

- `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.

8.7 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.

8.8 Catalyst Cooperative Code of Conduct

8.8.1 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.

8.8.2 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

8.8.3 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.

8.8.4 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.

8.8.5 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.

8.8.6 Attribution

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

8.9 pudl

8.9.1 pudl package

Subpackages

pudl.analysis package

Submodules

pudl.analysis.allocate_net_gen module

Allocated data from generation_fuel_eia923 table to generator level.

Net generation and fuel consumption is reported in two seperate tables in EIA 923: in the generation_eia923 and generation_fuel_eia923 tables. While the generation_fuel_eia923 table is more complete (the generation_eia923 table includes only ~55% of the reported MWhs), the generation_eia923 table is more granular (it is reported at the generator level).

This module allocates net generation and fuel consumption from the generation_fuel_eia923 table to the generator level. The main function here is `allocate_gen_fuel_by_gen()`.

The methodology we are employing here to allocate the net generation from the generation_fuel_eia923 table is not the only option and includes many assumptions. Firstly, this methodology assumes the generation_fuel_eia923 table is the ground truth for net generation - as opposed to the generation_eia923 table. We are making this assumption because we know that the generation_fuel_eia923 table is necessarily more complete - there are many full plants or generators in plants that do not report to the generation_eia923 table at all.

The next important note is the way in which we associated the data reported in the generation_fuel_eia923 table with generators. The generation_fuel_eia923 table is reported at the level of prime_mover_code/fuel_type (See `IDX_PM_FUEL`). Generators have prime_mover_codes, fuel_types (in energy_source_code_*s) and report_dates. This methology does not distinguish between primary and secondary fuel_types for generators - it associates portions of net generatoion to each prime_mover_code/fuel_type.

The last high-level point about this methodology surrounds the allocation method. In order to allocate portions of the net generation, we calculate as allocation ratio, which is based on the net generation from the generation_eia923 table when available and the capacity_mw from the generators_eia860 table. Some plants have a portion of their generators that report to generation_eia923. For those plants, we assign an allocation ratio in three steps: first we generate an allocation ratio based on capacity_mw for each group of generators (generators the do report in generation_eia923 and those that do not). Then we generate an allocation ratio based on the net generation reported in generation_eia923. Then we multiply both allocation ratios together to scale down the net generation based ratio based on the capacity of the generators reporting in generation_eia923.

This methodology has several potentail flaws and drawbacks. Because there is no indicator of what portion of the energy_source_codes (ie. fule_type), we associate the net generation equally amoung them. In effect, if a plant had multiple generators with the same prime_mover_code but opposite primary and secondary fuels (eg. gen 1 has a primary fuel of 'NG' and secondard fuel of 'DFO', while gen 2 has a primary fuel of 'DFO' and a secondary fuel of 'NG'), the methodology associates the generation_fuel_eia923 records similarly across these two generators. Nonetheless, the allocated net generation will still be porporational to each generators generation_eia923 net generation or capacity.

This methodology also has an effect of smoothing differences of generators with the same prime_mover_code and fuel_type. In effect, two similar generators will appear to have similar capacity factors, especially if they reported no data to the generation_eia923 table.

Another methodology that could be worth employing is use the generation_eia923 table when available and allocate the remaining net generation in a similar methodology as we have currently employed by using each generators' capacity as an allocator. For the ~.2% of records which report more net generation in the generation_eia923 table, we would have to augment that methodology.

`pudl.analysis.allocate_net_gen.`**`DATA_COLS = ['net_generation_mwh', 'fuel_consumed_mmbtu']`**

Data columns from generation_fuel_eia923 that are being allocated.

`pudl.analysis.allocate_net_gen.`**`IDX_GENS = ['plant_id_eia', 'generator_id', 'report_date']`**

Id columns for generators.

`pudl.analysis.allocate_net_gen.`**`IDX_PM_FUEL = ['plant_id_eia', 'prime_mover_code', 'fuel_typ`**

Id columns for plant, prime mover & fuel type records.

`pudl.analysis.allocate_net_gen.`**`agg_by_generator`**(*gen_pm_fuel*, *pudl_out*)

Aggreate the allocated gen fuel data to the generator level.

Parameters **`gen_pm_fuel`** (`pandas.DataFrame`) – result of *allocate_gen_fuel_by_gen_pm_fuel()*

`pudl.analysis.allocate_net_gen.`**`allocate_gen_fuel_by_gen`**(*pudl_out*)

Allocate gen fuel data columns to generators.

The generation_fuel_eia923 table includes net generation and fuel consumption data at the plant/fuel type/prime mover level. The most granular level of plants that PUDL typically uses is at the plant/generator level. This method converts the generation_fuel_eia923 table to the level of plant/generators.

Parameters **`pudl_out`** (`pudl.output.pudltabl.PudlTabl`) – An object used to create the tables for EIA and FERC Form 1 analysis.

Returns table with columns `IDX_GENS` and `DATA_COLS`. The `DATA_COLS` will be scaled to the level of the `IDX_GENS`.

Return type pandas.DataFrame

`pudl.analysis.allocate_net_gen.`**`allocate_gen_fuel_by_gen_pm_fuel`**(*pudl_out*)

Proportionally allocate net gen from gen_fuel table to generators.

Two main steps here:

- associated gen_fuel data w/ generators
- allocate gen_fuel data proportionally

The assocation process happens via *associate_gen_tables()*.

The allocation process entails generating a ratio for each record within a `IDX_PM_FUEL` group. We have two options for generating this ratio: the net generation in the generation_eia923 table and the capacity from the generators_eia860 table. We calculate both these ratios, then used the net generation based ratio if available to allocation a portion of the associated data fields.

Parameters **pudl_out** (`pudl.output.pudltabl.PudlTabl`) – An object used to create the tables for EIA and FERC Form 1 analysis.

Returns pandas.DataFrame

`pudl.analysis.allocate_net_gen.`**`associate_gen_tables`**(*pudl_out*)

Assocaite the three tables needed to assign net gen to generators.

Parameters **pudl_out** (`pudl.output.pudltabl.PudlTabl`) – An object used to create the tables for EIA and FERC Form 1 analysis.

`pudl.analysis.allocate_net_gen.`**`make_allocation_ratio`**(*gens_asst*)

Generate a ratio to use to allocate net generation by.

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 overal 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 associtated 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.
- **`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.
- **`max_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

pudl.analysis.service_territory module

Compile historical utility and balancing area territories.

Use the mapping of utilities to counties, and balancing areas to utilities, available within the EIA 861, in conjunction with the US Census geometries for counties, to infer the historical spatial extent of utility and balancing area territories. Output the resulting geometries for use in other applications.

`pudl.analysis.service_territory.`**`add_geometries`**(*df*, *census_gdf*, *dissolve=False*, *dissolve_by=None*)

Merge census geometries into dataframe on county_id_fips, optionally dissolving.

Merge the US Census county-level geospatial information into the DataFrame df based on the the column county_id_fips (in df), which corresponds to the column GEOID10 in census_gdf. Also bring in the population and area of the counties, summing as necessary in the case of dissolved geometries.

Parameters

- **`df`** (*`pandas.DataFrame`*) – A DataFrame containing a county_id_fips column.
- **`census_gdf`** (*`geopandas.GeoDataFrame`*) – A GeoDataFrame based on the US Census demographic profile (DP1) data at county resolution, with the original column names as published by US Census.
- **`dissolve`** (*`bool`*) – If True, dissolve individual county geometries into larger service territories.
- **`dissolve_by`** (*`list`*) – The columns to group by in the dissolve. For example, dissolve_by=["report_date", "utility_id_eia"] might provide annual utility service territories, while ["report_date", "balancing_authority_id_eia"] would provide annual balancing authority territories.

Returns geopandas.GeoDataFrame

`pudl.analysis.service_territory.`**`compile_geoms`**(*pudl_out*, *census_counties*, *entity_type*, *dissolve=False*, *limit_by_state=True*, *save=True*)

Compile all available utility or balancing authority geometries.

Parameters

- **`pudl_out`** (`pudl.output.pudltabl.PudlTabl`) – A PUDL output object, which will be used to extract and cache the EIA 861 tables.
- **`census_counties`** (*`geopandas.GeoDataFrame`*) – A GeoDataFrame containing the county level US Census DP1 data and county geometries.
- **`entity_type`** (*`str`*) – The type of service territory geometry to compile. Must be either "ba" (balancing authority) or "util" (utility).
- **`dissolve`** (*`bool`*) – Whether to dissolve the compiled geometries to the utility/balancing authority level, or leave them as counties.
- **`limit_by_state`** (*`bool`*) – Whether to limit included counties to those with observed EIA 861 data in association with the state and utility/balancing authority.
- **`save`** (*`bool`*) – If True, save the compiled GeoDataFrame as a GeoParquet file before returning. Especially useful in the case of dissolved geometries, as they are computationally expensive.

Returns geopandas.GeoDataFrame

`pudl.analysis.service_territory.`**`get_all_utils`**(*pudl_out*)

Compile IDs and Names of all known EIA Utilities.

Grab all EIA utility names and IDs from both the EIA 861 Service Territory table and the EIA 860 Utility entity table. This is a temporary function that's only needed because we haven't integrated the EIA 861 information into the entity harvesting process and PUDL database yet.

Parameters **`pudl_out`** (`pudl.output.pudltabl.PudlTabl`) – The PUDL output object which should be used to obtain PUDL data.

Returns Having 2 columns `utility_id_eia` and `utility_name_eia`.

Return type pandas.DataFrame

`pudl.analysis.service_territory.`**`get_territory_fips`**(*ids*, *assn*, *assn_col*, *st_eia861*, *limit_by_state=True*)

Compile county FIPS codes associated with an entity's service territory.

For each entity identified by ids, look up the set of counties associated with that entity on an annual basis. Optionally limit the set of counties to those within states where the selected entities reported activity elsewhere within the EIA 861 data.

Parameters

- **`ids`** (*`iterable of ints`*) – A collection of EIA utility or balancing authority IDs.
- **`assn`** (*`pandas.DataFrame`*) – Association table, relating `report_date`,
- **`state`** – column indicated by `assn_col` – if it's not `utility_id_eia`.
- **`utility_id_eia to each other`** (*`and`*) – column indicated by `assn_col` – if it's not `utility_id_eia`.
- **`well as the`** (*`as`*) – column indicated by `assn_col` – if it's not `utility_id_eia`.
- **`assn_col`** (*`str`*) – Label of the dataframe column in `assn` that contains the ID of the entities of interest. Should probably be either `balancing_authority_id_eia` or `utility_id_eia`.
- **`st_eia861`** (*`pandas.DataFrame`*) – The EIA 861 Service Territory table.
- **`limit_by_state`** (*`bool`*) – Whether to require that the counties associated with the balancing authority are inside a state that has also been seen in association with the balancing authority and the utility whose service territory contians the county.

Returns A table associating the entity IDs with a collection of counties annually, identifying counties both by name and county_id_fips (both state and state_id_fips are included for clarity).

Return type pandas.DataFrame

`pudl.analysis.service_territory.`**`get_territory_geometries`**(*ids*, *assn*, *assn_col*, *st_eia861*, *census_gdf*, *limit_by_state=True*, *dissolve=False*)

Compile service territory geometries based on county_id_fips.

Calls `get_territory_fips` to generate the list of counties associated with each entity identified by `ids`, and then merges in the corresponding county geometries from the US Census DP1 data passed in via `census_gdf`.

Optionally dissolve all of the county level geometries into a single geometry for each combination of entity and year.

Note: Dissolving geometires is a costly operation, and may take half an hour or more if you are processing all entities for all years. Dissolving also means that all the per-county information will be lost, rendering the output inappropriate for use in many analyses. Dissolving is mostly useful for generating visualizations.

Parameters

- **ids** (*iterable of ints*) – A collection of EIA balancing authority IDs.
- **assn** (*pandas.DataFrame*) – Association table, relating `report_date`,
- **state** – column indicated by `assn_col` – if it's not `utility_id_eia`.
- **utility_id_eia to each other** (*and*) – column indicated by `assn_col` – if it's not `utility_id_eia`.
- **well as the** (*as*) – column indicated by `assn_col` – if it's not `utility_id_eia`.
- **assn_col** (*str*) – Label of the dataframe column in `assn` that contains the ID of the entities of interest. Should probably be either `balancing_authority_id_eia` or `utility_id_eia`.
- **st_eia861** (*pandas.DataFrame*) – The EIA 861 Service Territory table.
- **census_gdf** (*geopandas.GeoDataFrame*) – The US Census DP1 county-level geometries as returned by pudl.output.censusdp1tract.get_layer("county").
- **limit_by_state** (*bool*) – Whether to require that the counties associated with the balancing authority are inside a state that has also been seen in association with the balancing authority and the utility whose service territory contians the county.
- **dissolve** (*bool*) – If False, each record in the compiled territory will correspond to a single county, with a county-level geometry, and there will be many records enumerating all the counties associated with a given balancing_authority_id_eia in each year. If dissolve=True, all of the county-level geometries for each utility in each year will be merged together ("dissolved") resulting in a single geometry and record for each balancing_authority-year.

Returns geopandas.GeoDataFrame

`pudl.analysis.service_territory.`**`main`**`()`

Compile historical utility and balancing area territories.

`pudl.analysis.service_territory.`**`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.analysis.service_territory.`**`plot_all_territories`**`(`*gdf*, *report_date*, *respondent_type=('balancing_authority', 'utility')*, *color='black'*, *alpha=0.25*, *basemap=True*`)`

Plot all of the planning areas of a given type for a given report date.

Todo: This function needs to be made more general purpose, and less entangled with the FERC 714 data.

Parameters

- **gdf** (*geopandas.GeoDataFrame*) – GeoDataFrame containing planning area geometries, organized by respondent_id_ferc714 and report_date.
- **report_date** (*datetime*) – A Datetime indicating what year's planning areas should be displayed.
- **respondent_type** (*str or iterable*) – Type of respondent whose planning areas should be displayed. Either "utility" or "balancing_authority" or an iterable collection containing both.
- **color** (*str*) – Color to use for the planning areas.
- **alpha** (*float*) – Transparency to use for the planning areas.
- **basemap** (*bool*) – If true, use the OpenStreetMap tiles for context.

Returns matplotlib.axes.Axes

pudl.analysis.service_territory.**plot_historical_territory**(*gdf*, *id_col*, *id_val*)

Plot all the historical geometries defined for the specified entity.

This is useful for exploring how a particular entity's service territory has evolved over time, or for identifying individual missing or inaccurate territories.

Parameters

- **gdf** (*geopandas.GeoDataFrame*) – A geodataframe containing geometries pertaining electricity planning areas. Can be broken down by county FIPS code, or have a single record containing a geometry for each combination of report_date and the column being used to select planning areas (see below).
- **id_col** (*str*) – The label of a column in gdf that identifies the planning area to be visualized, like utility_id_eia, balancing_authority_id_eia, or balancing_authority_code_eia.
- **id_val** (*str or int*) – The value identifying the

Returns None

pudl.analysis.timeseries_cleaning module

Screen timeseries for anomalies and impute missing and anomalous values.

The screening methods were originally designed to identify unrealistic data in the electricity demand timeseries reported to EIA on Form 930, and have also been applied to the FERC Form 714, and various historical demand timeseries published by regional grid operators like MISO, PJM, ERCOT, and SPP.

They are adapted from code published and modified by: * Tyler Ruggles <truggles@carnegiescience.edu> * Greg Schivley <greg@carbonimpact.co>

And described at: * https://doi.org/10.1038/s41597-020-0483-x * https://zenodo.org/record/3737085 * https://github.com/truggles/EIA_Cleaned_Hourly_Electricity_Demand_Code

The imputation methods were designed for multivariate time series forecasting.

They are adapted from code published by: * Xinyu Chen <chenxy346@gmail.com>

And described at: * https://arxiv.org/abs/2006.10436 * https://arxiv.org/abs/2008.03194 * https://github.com/xinychen/tensor-learning

class pudl.analysis.timeseries_cleaning.**Timeseries**(*x: Union[numpy.ndarray, pandas.core.frame.DataFrame]*)

Bases: object

Multivariate timeseries for anomalies detection and imputation.

xi
Reference to the original values (can be null). Many methods assume that these represent chronological, regular timeseries.

x
Copy of *xi* with any flagged values replaced with null.

flags
Flag label for each value, or null if not flagged.

flagged
Running list of flags that have been checked so far.

index
Row index.

columns
Column names.

diff (*shift: int = 1*) → numpy.ndarray
Values minus the value of their neighbor.

Parameters shift – Positions to shift for calculating the difference. Positive values select a preceding (left) neighbor.

flag (*mask: numpy.ndarray*, *flag: str*) → None
Flag values.

Flags values (if not already flagged) and nulls flagged values.

Parameters

- **mask** – Boolean mask of the values to flag.
- **flag** – Flag name.

flag_anomalous_region (*window: int = 48*, *threshold: float = 0.15*) → None
Flag values surrounded by flagged values (ANOMALOUS_REGION).

Original null values are not considered flagged values.

Parameters

- **width** – Width of regions.
- **threshold** – Fraction of flagged values required for a region to be flagged.

flag_double_delta (*iqr_window: int = 240*, *multiplier: float = 2*) → None
Flag values very different from their neighbors on either side (DOUBLE_DELTA).

Flags values whose differences to both neighbors on either side exceeds a *multiplier* times the rolling interquartile range (IQR) of neighbor difference.

Parameters

- **iqr_window** – Number of values in the moving window for the rolling IQR of neighbor difference.
- **multiplier** – Number of times the rolling IQR of neighbor difference the value's difference to its neighbors must exceed for the value to be flagged.

flag_global_outlier (*medians: float = 9*) → None
Flag values greater or less than n times the global median (GLOBAL_OUTLIER).

Parameters **medians** – Number of times the median the value must exceed the median.

flag_global_outlier_neighbor(*neighbors: int = 1*) → None
Flag values neighboring global outliers (GLOBAL_OUTLIER_NEIGHBOR).

Parameters **neighbors** – Number of neighbors to flag on either side of each outlier.

Raises **ValueError** – Global outliers must be flagged first.

flag_identical_run(*length: int = 3*) → None
Flag the last values in identical runs (IDENTICAL_RUN).

Parameters **length** – Run length to flag. If *3*, the third (and subsequent) identical values are flagged.

Raises **ValueError** – Run length must be 2 or greater.

flag_local_outlier(*window: int = 48, shifts: Sequence[int] = range(- 240, 241, 24), long_window: int = 480, iqr_window: int = 240, multiplier: Tuple[float, float] = (3.5, 2.5)*) → None
Flag local outliers (LOCAL_OUTLIER_HIGH, LOCAL_OUTLIER_LOW).

Flags values which are above or below the *median_prediction()* by more than a *multiplier* times the *rolling_iqr_of_rolling_median_offset()*.

Parameters

- **window** – Number of values in the moving window for the local rolling median.
- **shifts** – Positions to shift the local rolling median offset by, for computing its median.
- **long_window** – Number of values in the moving window for the regional (long) rolling median.
- **iqr_window** – Number of values in the moving window for the rolling interquartile range (IQR).
- **multiplier** – Number of times the *rolling_iqr_of_rolling_median_offset()* the value must be above (HIGH) and below (LOW) the *median_prediction()* to be flagged.

flag_negative_or_zero() → None
Flag negative or zero values (NEGATIVE_OR_ZERO).

flag_ruggles() → None
Flag values following the method of Ruggles and others (2020).

Assumes values are hourly electricity demand.

- description: https://doi.org/10.1038/s41597-020-0483-x
- code: https://github.com/truggles/EIA_Cleaned_Hourly_Electricity_Demand_Code

flag_single_delta(*window: int = 48, shifts: Sequence[int] = range(- 240, 241, 24), long_window: int = 480, iqr_window: int = 240, multiplier: float = 5, rel_multiplier: float = 15*) → None
Flag values very different from the nearest unflagged value (SINGLE_DELTA).

Flags values whose difference to the nearest unflagged value, with respect to value and relative median prediction, differ by less than a multiplier times the rolling interquartile range (IQR) of the difference - *multiplier* times *rolling_iqr_of_diff()* and *rel_multiplier* times `iqr_of_diff_of_relative_mean_prediction()`, respectively.

Parameters

- **window** – Number of values in the moving window for the rolling median (for the relative median prediction).
- **shifts** – Positions to shift the local rolling median offset by, for computing its median (for the relative median prediction).
- **long_window** – Number of values in the moving window for the long rolling median (for the relative median prediction).
- **iqr_window** – Number of values in the moving window for the rolling IQR of neighbor difference.
- **multiplier** – Number of times the rolling IQR of neighbor difference the value's difference to its neighbor must exceed for the value to be flagged.
- **rel_multiplier** – Number of times the rolling IQR of relative median prediction the value's prediction difference to its neighbor must exceed for the value to be flagged.

fold_tensor(*x: Optional[numpy.ndarray] = None*, *periods: int = 24*) → numpy.ndarray

Fold into a 3-dimensional tensor representation.

Folds the series *x* (number of observations, number of series) into a 3-d tensor (number of series, number of groups, number of periods), splitting observations into groups of length *periods*. For example, each group may represent a day and each period the hour of the day.

Parameters

- **x** – Series array to fold. Uses *x* by default.
- **periods** – Number of consecutive values in each series to fold into a group.

Returns

```
>>> x = np.column_stack([[1, 2, 3, 4, 5, 6], [10, 20, 30, 40, 50, 
↪60]])
>>> s = Timeseries(x)
>>> tensor = s.fold_tensor(periods=3)
>>> tensor[0]
array([[1, 2, 3],
       [4, 5, 6]])
>>> np.all(x == s.unfold_tensor(tensor))
True
```

impute(*mask: Optional[numpy.ndarray] = None*, *periods: int = 24*, *blocks: int = 1*, *method: str = 'tubal'*, ***kwargs: Any*) → numpy.ndarray

Impute null values.

Note: The imputation method requires that nulls be replaced by zeros, so the series cannot already contain zeros.

Parameters

- **mask** – Boolean mask of values to impute in addition to any null values in *x*.
- **periods** – Number of consecutive values in each series to fold into a group. See `fold_tensor()`.
- **blocks** – Number of blocks into which to split the series for imputation. This has been found to reduce processing time for *method='tnn'*.

- **method** – Imputation method to use ('tubal': `impute_latc_tubal()`, 'tnn': `impute_latc_tnn()`).
- **kwargs** – Optional arguments to *method*.

Returns Array of same shape as `x` with all null values (and those selected by *mask*) replaced with imputed values.

Raises `ValueError` – Zero values present. Replace with very small value.

iqr_of_diff_of_relative_median_prediction(*shift: int = 1*, ***kwargs: Any*) → numpy.ndarray

Interquartile range of the running difference of the relative median prediction.

Parameters

- **shift** – Positions to shift for calculating the difference. Positive values select a preceding (left) neighbor.
- **kwargs** – Arguments to `relative_median_prediction()`.

median_of_rolling_median_offset(*window: int = 48*, *shifts: Sequence[int] = range(- 240, 241, 24)*) → numpy.ndarray

Median of the offset from the rolling median.

Calculated by shifting the rolling median offset (`rolling_median_offset()`) by different numbers of values, then taking the median at each position. Estimates the typical local cycle in cyclical data.

Parameters

- **window** – Number of values in the moving window for the rolling median.
- **shifts** – Number of values to shift the rolling median offset by.

median_prediction(*window: int = 48*, *shifts: Sequence[int] = range(- 240, 241, 24)*, *long_window: int = 480*) → numpy.ndarray

Values predicted from local and regional rolling medians.

Calculated as *{ local median } + { median of local median offset } * { local median } / { regional median }*.

Parameters

- **window** – Number of values in the moving window for the local rolling median.
- **shifts** – Positions to shift the local rolling median offset by, for computing its median.
- **long_window** – Number of values in the moving window for the regional (long) rolling median.

plot_flags(*name: Any = 0*) → None

Plot cleaned series and anomalous values colored by flag.

Parameters **name** – Series to plot, as either an integer index or name in `columns`.

relative_median_prediction(***kwargs: Any*) → numpy.ndarray

Values divided by their value predicted from medians.

Parameters **kwargs** – Arguments to `median_prediction()`.

rolling_iqr_of_diff(*shift: int = 1*, *window: int = 240*) → numpy.ndarray

Rolling interquartile range (IQR) of the difference between neighboring values.

Parameters

- **shift** – Positions to shift for calculating the difference.
- **window** – Number of values in the moving window for the rolling IQR.

rolling_iqr_of_rolling_median_offset(*window: int = 48*, *iqr_window: int = 240*) → numpy.ndarray
Rolling interquartile range (IQR) of rolling median offset.

Estimates the spread of the local cycles in cyclical data.

Parameters

- **window** – Number of values in the moving window for the rolling median.
- **iqr_window** – Number of values in the moving window for the rolling IQR.

rolling_median(*window: int = 48*) → numpy.ndarray
Rolling median of values.

Parameters **window** – Number of values in the moving window.

rolling_median_offset(*window: int = 48*) → numpy.ndarray
Values minus the rolling median.

Estimates the local cycle in cyclical data by removing longterm trends.

Parameters **window** – Number of values in the moving window.

simulate_nulls(*lengths: Optional[Sequence[int]] = None*, *padding: int = 1*, *intersect: bool = False*, *overlap: bool = False*) → numpy.ndarray
Find non-null values to null to match a run-length distribution.

Parameters

- **length** – Length of null runs to simulate for each series. By default, uses the run lengths of null values in each series.
- **padding** – Minimum number of non-null values between simulated null runs and between simulated and existing null runs.
- **intersect** – Whether simulated null runs can intersect each other.
- **overlap** – Whether simulated null runs can overlap existing null runs. If *True*, *padding* is ignored.

Returns Boolean mask of current non-null values to set to null.

Raises **ValueError** – Cound not find space for run of length {length}.

Examples

```
>>> x = np.column_stack([[1, 2, np.nan, 4, 5, 6, 7, np.nan, np.nan]])
>>> s = Timeseries(x)
>>> s.simulate_nulls().ravel()
array([ True, False, False, False, True, True, False, False, False])
>>> s.simulate_nulls(lengths=[4], padding=0).ravel()
array([False, False, False, True, True, True, True, False, False])
```

summarize_flags() → pandas.core.frame.DataFrame
Summarize flagged values by flag, count and median.

summarize_imputed(*imputed: numpy.ndarray*, *mask: numpy.ndarray*) → pandas.core.frame.DataFrame
Summarize the fit of imputed values to actual values.

Summarizes the agreement between actual and imputed values with the following statistics:

- *mpe*: Mean percent error, *(actual - imputed) / actual*.

- *mape*: Mean absolute percent error, *abs(mpe)*.

Parameters

- **imputed** – Series of same shape as *x* with imputed values. See *impute()*.
- **mask** – Boolean mask of imputed values that were not null in *x*. See *simulate_nulls()*.

Returns Table of imputed value statistics for each series.

unflag (*flags: Iterable[str]*) → None
Unflag values.

Unflags values by restoring their original values and removing their flag.

Parameters **flags** – Flag names.

unfold_tensor (*tensor: numpy.ndarray*) → numpy.ndarray
Unfold a 3-dimensional tensor representation.

Performs the reverse of *fold_tensor()*.

pudl.analysis.timeseries_cleaning.**array_diff** (*x: numpy.ndarray, periods: int = 1, axis: int = 0, fill: Any = nan*) → numpy.ndarray
First discrete difference of array elements.

This is a fast numpy implementation of `pd.DataFrame.diff()`.

Parameters

- **periods** – Periods to shift for calculating difference, accepts negative values.
- **axis** – Array axis along which to calculate the difference.
- **fill** – Value to use at the margins where a difference cannot be calculated.

Returns Array of same shape and type as *x* with discrete element differences.

Examples

```
>>> x = np.random.random((4, 2))
>>> np.all(array_diff(x, 1)[1:] == pd.DataFrame(x).diff(1).values[1:])
True
>>> np.all(array_diff(x, 2)[2:] == pd.DataFrame(x).diff(2).values[2:])
True
>>> np.all(array_diff(x, -1)[:-1] == pd.DataFrame(x).diff(-1).values[:-1])
True
```

pudl.analysis.timeseries_cleaning.**encode_run_length** (*x: Union[Sequence, numpy.ndarray]*) → Tuple[numpy.ndarray, numpy.ndarray]
Encode vector with run-length encoding.

Parameters **x** – Vector to encode.

Returns Values and their run lengths.

Examples

```
>>> x = np.array([0, 1, 1, 0, 1])
>>> encode_run_length(x)
(array([0, 1, 0, 1]), array([1, 2, 1, 1]))
>>> encode_run_length(x.astype('bool'))
(array([False,  True, False,  True]), array([1, 2, 1, 1]))
>>> encode_run_length(x.astype('<U1'))
(array(['0', '1', '0', '1'], dtype='<U1'), array([1, 2, 1, 1]))
>>> encode_run_length(np.where(x == 0, np.nan, x))
(array([nan,  1., nan,  1.]), array([1, 2, 1, 1]))
```

`pudl.analysis.timeseries_cleaning.`**`impute_latc_tnn`**(*tensor: numpy.ndarray, lags: Sequence[int] = [1], alpha: Sequence[float] = [0.3333333333333333, 0.3333333333333333, 0.3333333333333333], rho0: float = 1e-07, lambda0: float = 2e-07, theta: int = 20, epsilon: float = 1e-07, maxiter: int = 300*) → numpy.ndarray

Impute tensor values with LATC-TNN method by Chen and Sun (2020).

Uses low-rank autoregressive tensor completion (LATC) with truncated nuclear norm (TNN) minimization.

- description: https://arxiv.org/abs/2006.10436
- code: https://github.com/xinychen/tensor-learning/blob/master/mats

Parameters

- **`tensor`** – Observational series in the form (series, groups, periods). Null values are replaced with zeros, so any zeros will be treated as null.
- **`lags`** –
- **`alpha`** –
- **`rho0`** –
- **`lambda0`** –
- **`theta`** –
- **`epsilon`** – Convergence criterion. A smaller number will result in more iterations.
- **`maxiter`** – Maximum number of iterations.

Returns Tensor with missing values in *tensor* replaced by imputed values.

`pudl.analysis.timeseries_cleaning.`**`impute_latc_tubal`**(*tensor: numpy.ndarray, lags: Sequence[int] = [1], rho0: float = 1e-07, lambda0: float = 2e-07, epsilon: float = 1e-07, maxiter: int = 300*) → numpy.ndarray

Impute tensor values with LATC-Tubal method by Chen, Chen and Sun (2020).

Uses low-tubal-rank autoregressive tensor completion (LATC-Tubal). It is much faster than `impute_latc_tnn()` for very large datasets, with comparable accuracy.

- description: https://arxiv.org/abs/2008.03194

- code: https://github.com/xinychen/tensor-learning/blob/master/mats

Parameters

- **tensor** – Observational series in the form (series, groups, periods). Null values are replaced with zeros, so any zeros will be treated as null.
- **lags** –
- **rho0** –
- **lambda0** –
- **epsilon** – Convergence criterion. A smaller number will result in more iterations.
- **maxiter** – Maximum number of iterations.

Returns Tensor with missing values in *tensor* replaced by imputed values.

`pudl.analysis.timeseries_cleaning.`**`insert_run_length`**(*x: Union[Sequence, numpy.ndarray], values: Union[Sequence, numpy.ndarray], lengths: Sequence[int], mask: Optional[Sequence[bool]] = None, padding: int = 0, intersect: bool = False*) → numpy.ndarray

Insert run-length encoded values into a vector.

Parameters

- **x** – Vector to insert values into.
- **values** – Values to insert.
- **lengths** – Length of run to insert for each value in *values*.
- **mask** – Boolean mask, of the same length as *x*, where values can be inserted. By default, values can be inserted anywhere in *x*.
- **padding** – Minimum space between inserted runs and, if *mask* is provided, the edges of masked-out areas.
- **intersect** – Whether to allow inserted runs to intersect each other.

Raises

- **ValueError** – Padding must zero or greater.
- **ValueError** – Run length must be greater than zero.
- **ValueError** – Cound not find space for run of length {length}.

Returns Copy of array *x* with values inserted.

Example

```
>>> x = [0, 0, 0, 0]
>>> mask = [True, False, True, True]
>>> insert_run_length(x, values=[1, 2], lengths=[1, 2], mask=mask)
array([1, 0, 2, 2])
```

If we use unique values for the background and each inserted run, the run length encoding of the result (ignoring the background) is the same as the inserted run, albeit in a different order.

```
>>> x = np.zeros(10, dtype=int)
>>> values = [1, 2, 3]
>>> lengths = [1, 2, 3]
>>> x = insert_run_length(x, values=values, lengths=lengths)
>>> rvalues, rlengths = encode_run_length(x[x != 0])
>>> order = np.argsort(rvalues)
>>> all(rvalues[order] == values) and all(rlengths[order] == lengths)
True
```

Null values can be inserted into a vector such that the new null runs match the run length encoding of the existing null runs.

```
>>> x = [1, 2, np.nan, np.nan, 5, 6, 7, 8, np.nan]
>>> is_nan = np.isnan(x)
>>> rvalues, rlengths = encode_run_length(is_nan)
>>> xi = insert_run_length(
...     x,
...     values=[np.nan] * rvalues.sum(),
...     lengths=rlengths[rvalues],
...     mask=~is_nan
... )
>>> np.isnan(xi).sum() == 2 * is_nan.sum()
True
```

The same as above, with non-zero *padding*, yields a unique solution:

```
>>> insert_run_length(
...     x,
...     values=[np.nan] * rvalues.sum(),
...     lengths=rlengths[rvalues],
...     mask=~is_nan,
...     padding=1
... )
array([nan,  2., nan, nan,  5., nan, nan,  8., nan])
```

`pudl.analysis.timeseries_cleaning.`**`slice_axis`**(*x: numpy.ndarray*, *start: Optional[int] = None*, *end: Optional[int] = None*, *step: Optional[int] = None*, *axis: int = 0*) → Tuple

Return an index that slices an array along an axis.

Parameters

- **x** – Array to slice.
- **start** – Start index of slice.
- **end** – End index of slice.
- **step** – Step size of slice.

- **axis** – Axis along which to slice.

Returns Tuple of `slice` that slices array *x* along axis *axis* (*x[..., start:stop:step]*).

Examples

```
>>> x = np.random.random((3, 4, 5))
>>> np.all(x[1:] == x[slice_axis(x, start=1, axis=0)])
True
>>> np.all(x[:, 1:] == x[slice_axis(x, start=1, axis=1)])
True
>>> np.all(x[:, :, 1:] == x[slice_axis(x, start=1, axis=2)])
True
```

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.censusdp1tract_to_sqlite module

Convert the US Census DP1 ESRI GeoDatabase into an SQLite Database.

This is a thin wrapper around the GDAL ogr2ogr command line tool. We use it to convert the Census DP1 data which is distributed as an ESRI GeoDB into an SQLite DB. The module provides ogr2ogr with the Census DP 1 data from the PUDL datastore, and directs it to be output into the user's SQLite directory alongside our other SQLite Databases (ferc1.sqlite and pudl.sqlite)

Note that the ogr2ogr command line utility must be available on the user's system for this to work. This tool is part of the `pudl-dev` conda environment, but if you are using PUDL outside of the conda environment, you will need to install ogr2ogr separately. On Debian Linux based systems such as Ubuntu it can be installed with `sudo apt-get install gdal-bin` (which is what we do in our CI setup and Docker images.)

`pudl.convert.censusdp1tract_to_sqlite.`**`censusdp1tract_to_sqlite`**(*pudl_settings=None, year=2010*)

Use GDAL's ogr2ogr utility to convert the Census DP1 GeoDB to an SQLite DB.

The Census DP1 GeoDB is read from the datastore, where it is stored as a zipped archive. This archive is unzipped into a temporary directory so that ogr2ogr can operate on the ESRI GeoDB, and convert it to SQLite. The resulting SQLite DB file is put in the PUDL output directory alongside the ferc1 and pudl SQLite databases.

Parameters

- **pudl_settings** (*dict*) – A PUDL settings dictionary.
- **year** (*int*) – Year of Census data to extract (currently must be 2010)

Returns None

`pudl.convert.censusdp1tract_to_sqlite.`**`main`**`()`

Convert the Census DP1 GeoDatabase into an SQLite Database.

`pudl.convert.censusdp1tract_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_rst module

Module to convert json metadata into rst files.

All of the information about the transformed pudl tables, namely their fields types and descriptions, resides in the datapackage metadata. This module makes that information available to users, without duplicating any data, by converting json metadata files into documentation-compatible rst files. The functions serve to extract the field names, field data types, and field descriptions of each pudl table and outputs them in a manner that automatically updates the read-the-docs.

`pudl.convert.datapkg_to_rst.`**`RST_TEMPLATE = '\n==============================================`

A template to map data from a json dictionary into one rst file. Contains multiple tables seperated by headers.

`pudl.convert.datapkg_to_rst.`**`datapkg2rst`**`(`*meta_json*, *meta_rst*, *ignore=None*`)`

Convert json metadata to a single rst file.

`pudl.convert.datapkg_to_rst.`**`logger = <Logger pudl.convert.datapkg_to_rst (WARNING)>`**

The following templates map json data into one long rst file seperated by table titles and document links (RST_TEMPLATE)

It's important for the templates that the json data do not contain excess white space either at the beginning or the end of each value.

`pudl.convert.datapkg_to_rst.`**`main`**`()`

Run conversion from json to rst.

`pudl.convert.datapkg_to_rst.`**`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 module

Merge compatible PUDL datapackages and load the result into an SQLite DB.

This script merges a set of compatible PUDL datapackages into a single tabular datapackage, and then loads that package into the PUDL SQLite DB

The input datapackages must all have been produced in the same ETL run, and share the same `datapkg-bundle-uuid` value. Any data sources (e.g. ferc1, eia923) that appear in more than one of the datapackages to be merged must also share identical ETL parameters (years, tables, states, etc.), allowing easy deduplication of resources.

Having the ability to load only a subset of the datapackages resulting from an ETL run into the SQLite database is helpful because larger datasets are much easier to work with via columnar datastores like Apache Parquet – loading all of EPA CEMS into SQLite can take more than 24 hours. PUDL also provides a separate epacems_to_parquet script that can be used to generate a Parquet dataset that is partitioned by state and year, which can be read directly into pandas or dask dataframes, for use in conjunction with the other PUDL data that is stored in the SQLite DB.

`pudl.convert.datapkg_to_sqlite.`**`datapkg_to_sqlite`**(*sqlite_url*, *out_path*, *clobber=False*, *fkeys=False*)

Load a PUDL datapackage into a sqlite database.

Parameters

- **`sqlite_url`** (*`str`*) – An SQLite database connection URL.
- **`out_path`** (*`path-like`*) – Path to the base directory of the datapackage to be loaded into SQLite. Must contain the datapackage.json file.
- **`clobber`** (*`bool`*) – If True, replace an existing PUDL DB if it exists. If False (the default), fail if an existing PUDL DB is found.
- **`fkeys`** (*`bool`*) – If true, tell SQLite to check foreign key constraints for the records that are being loaded. Left off by default.

Returns None

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

Merge PUDL datapackages and save them into an 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.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 SQLite 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.

`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.`**`create_in_dtypes`**`()`

Create a dictionary of input data types.

This specifies the dtypes of the input columns, which is necessary for some cases where, e.g., a column is always NaN.

Returns mapping columns names to `pandas` data types.

Return type dict

`pudl.convert.epacems_to_parquet.`**`epacems_to_parquet`**`(`*datapkg_path, epacems_years, epacems_states, out_dir, compression='snappy', partition_cols=('year', 'state'), clobber=False*`)`

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 datset on disk by year as well as state. (Year partitions follow the CEMS input data, based on local plant time. The operating_datetime_utc identifies time in UTC, so there's a mismatch of a few hours on December 31 / January 1.)

Parameters

- **`datapkg_path`** (*`path-like`*) – Path to the datapackage.json file describing the datapackage contaning the EPA CEMS data to be converted.
- **`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
- **`out_dir`** (*`path-like`*) – The directory in which to output the Parquet files
- **`compression`** (*`string`*) –
- **`partition_cols`** (*`tuple`*) –
- **`clobber`** (*`bool`*) – If True and there is already a directory with out_dirs name, the existing parquet files will be deleted and new ones will be generated in their place.

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.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.merge_datapkgs module

Functions for merging compatible PUDL datapackges together.

`pudl.convert.merge_datapkgs.check_etl_params(dps)`
Verify that datapackages to be merged have compatible ETL params.

Given that all of the input data packages come from the same ETL run, which means they will have used the same input data, the only way they should potentially differ is in the ETL parameters which were used to generate them. This function pulls the data source specific ETL params which we store in each datapackage descriptor and checks that within a given data source (e.g. eia923, ferc1) all of the ETL parameters are identical (e.g. the years, states, and tables loaded).

Parameters **dps** (*iterable*) – A list of datapackage.Package objects, representing the datapackages to be merged.

Returns None

Raises **ValueError** – If the PUDL ETL parameters associated with any given data source are not identical across all instances of that data source within the datapackages to be merged. Also if the ETL UUIDs for all of the datapackages to be merged are not identical.

`pudl.convert.merge_datapkgs.check_identical_vals(dps, required_vals, optional_vals=())`
Verify that datapackages to be merged have required identical values.

This only works for elements with simple (hashable) datatypes, which can be added to a set.

Parameters

- **dps** (*iterable*) – a list of tabular datapackage objects, output by PUDL.
- **required_vals** (*iterable*) – A list of strings indicating which top level metadata elements should be compared between the datapackages. All must be present in every datapackage.
- **optional_vals** (*iterable*) – A list of strings indicating top level metadata elements to be compared between the datapackages. They do not need to appear in all datapackages, but if they do appear, they must be identical.

Returns None

Raises

- **ValueError** – if any of the required or optional metadata elements have different values in the different data packages.
- **KeyError** – if a required metadata element is not found in any of the datapackages.

`pudl.convert.merge_datapkgs.`**`merge_data`**(*dps*, *out_path*)

Copy the CSV files into the merged datapackage's data directory.

Iterates through all of the resources in the input datapackages and copies the files they refer to into the data directory associated with the merged datapackage (a directory named "data" inside the out_path directory).

Function assumes that a fresh (empty) data directory has been created. If a file with the same name already exists, it is not overwritten, in order to prevent unnecessary copying of resources which appear in multiple input packages.

Parameters

- **dps** (*iterable*) – A list of datapackage.Package objects, representing the datapackages to be merged.
- **out_path** (*path like*) – Base directory for the newly created datapackage. The final path element will also be used as the name of the merged data package.

Returns None

`pudl.convert.merge_datapkgs.`**`merge_datapkgs`**(*dps*, *out_path*, *clobber=False*)

Merge several compatible datapackages into one larger datapackage.

Parameters

- **dps** (*iterable*) – A collection of tabular data package objects that were output by PUDL, to be merged into a single deduplicated datapackage for loading into a database or other storage medium.
- **out_path** (*path-like*) – Base directory for the newly created datapackage. The final path element will also be used as the name of the merged data package.
- **clobber** (*bool*) – If the location of the output datapackage already exists, should it be overwritten? If True, yes. If False, no.

Returns A report containing information about the validity of the merged datapackage.

Return type dict

Raises

- **FileNotFoundError** – If any of the input datapackage paths do not exist.
- **FileExistsError** – If the output directory exists and clobber is False.

`pudl.convert.merge_datapkgs.`**`merge_meta`**(*dps*, *datapkg_name*)

Merge the JSON descriptors of datapackages into one big descriptor.

This function builds up a new tabular datapackage JSON descriptor as a python dictionary, containing the merged metadata from all of the input datapackages.

The process is complex for two reasons. First, there are several different datatypes in the descriptor that need to be merged, and the processes for each of them are different. Second, what constitutes a "merge" may vary depending on the semantic content of the metadata. E.g. the `created` timestamp is a simple string, but we need to choose one of the several values (the earliest one) for inclusion in the merged datapackage, while many other simple string fields are required to be identical across all of the input data packages (e.g. `datapkg-bundle-uuid`):

Parameters

- **dps** (*iterable*) – A collection of datapackage objects, whose metadata will be merged to create a single datapackage descriptor representing the union of all the data in the input datapackages.
- **datapkg_name** (*str*) – The name associated with the newly merged datapackage. This should be the same as the name of the directory in which the datapackage is found.

Returns a Python dictionary representing a tabular datapackage JSON descriptor, encoded as a python dictionary, containing the merged metadata of the input datapackages.

Return type dict

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.ferc1_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.datapkg_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.

class `pudl.extract.eia860.`**`Extractor`**(**args*, ***kwargs*)

Bases: `pudl.extract.excel.GenericExtractor`

Extractor for the excel dataset EIA860.

static **`get_dtypes`**(*page*, ***partition*)

Returns dtypes for plant id columns.

`process_raw`(*df*, *page*, ***partition*)

Apply necessary pre-processing to the dataframe.

- Rename columns based on our compiled spreadsheet metadata
- Add report_year if it is missing
- Add a flag indicating if record came from EIA 860, or EIA 860M
- Fix any generator_id values with leading zeroes.

pudl.extract.eia860m module

Retrieve data from EIA Form 860M spreadsheets for analysis.

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

This code is for use analyzing EIA Form 860M data. EIA 860M is only used in conjunction with EIA 860. This module boths extracts EIA 860M and appends the extracted EIA 860M dataframes to the extracted EIA 860 dataframes. Example setup with pre-genrated *eia860_raw_dfs* and datastore as *ds*:

eia860m_raw_dfs = pudl.extract.eia860m.Extractor(ds).extract(pc.working_partitions['eia860m']['year_month'])

eia860_raw_dfs = pudl.extract.eia860m.append_eia860m(eia860_raw_dfs=eia860_raw_dfs, eia860m_raw_dfs=eia860m_raw_dfs)

class `pudl.extract.eia860m.`**`Extractor`**(**args*, ***kwargs*)

Bases: `pudl.extract.excel.GenericExtractor`

Extractor for the excel dataset EIA860M.

static get_dtypes(*page*, ***partition*)

Returns dtypes for plant id columns.

process_raw(*df*, *page*, ***partition*)

Adds source column and report_year column if missing.

`pudl.extract.eia860m.`**`append_eia860m`**(*eia860_raw_dfs*, *eia860m_raw_dfs*)

Append EIA 860M to the pages to.

Parameters

- **eia860_raw_dfs** (*dictionary*) – dictionary of pandas.Dataframe's from EIA 860 raw tables. Restult of pudl.extract.eia860.Extractor().extract()
- **eia860m_raw_dfs** (*dictionary*) – dictionary of pandas.Dataframe's from EIA 860M raw tables. Restult of pudl.extract.eia860m.Extractor().extract()

Returns augumented eia860_raw_dfs dictionary of pandas.DataFrame's. Each raw page stored in eia860m_raw_dfs appened to its eia860_raw_dfs counterpart.

Return type dictionary

pudl.extract.eia861 module

Retrieve data from EIA Form 861 spreadsheets for analysis.

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

This code is for use analyzing EIA Form 861 data.

class `pudl.extract.eia861.`**`Extractor`**(**args*, ***kwargs*)

Bases: `pudl.extract.excel.GenericExtractor`

Extractor for the excel dataset EIA861.

static get_dtypes(*page*, ***partition*)

Returns dtypes for plant id columns.

process_raw(*df*, *page*, ***partition*)

Rename columns with location.

static process_renamed(*df*, *page*, ***partition*)

Adds report_year column if missing.

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.

class pudl.extract.eia923.**Extractor**(**args*, ***kwargs*)
Bases: *pudl.extract.excel.GenericExtractor*

Extractor for EIA form 923.

static get_dtypes(*page*, ***partition*)
Returns dtypes for plant id columns.

static process_final_page(*df*, *page*)
Removes reserved columns from the final dataframe.

process_raw(*df*, *page*, ***partition*)
Drops reserved columns.

static process_renamed(*df*, *page*, ***partition*)
Cleans up unnamed_0 column in stocks page, drops invalid plan_id_eia rows.

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.**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

class pudl.extract.epacems.**EpaCemsDatastore**(*datastore: pudl.workspace.datastore.Datastore*)
Bases: object

Helper class to extract EpaCems resources from datastore.

EpaCems resources are identified by a year and a state. Each of these zip files contain monthly zip files that in turn contain csv files. This class implements get_data_frame method that will concatenate tables for a given state and month across all months.

get_data_frame(*partition: pudl.extract.epacems.EpaCemsPartition*) → pandas.core.frame.DataFrame
Constructs dataframe holding data for a given (year, state) partition.

class pudl.extract.epacems.**EpaCemsPartition**(*year: str*, *state: str*)
Bases: tuple

Represents EpaCems partition identifying unique resource file.

get_filters()
Returns filters for retrieving given partition resource from Datastore.

get_key()
Returns hashable key for use with EpaCemsDatastore.

get_monthly_file(*month: int*) → pathlib.Path
Returns the filename (without suffix) that contains the monthly data.

state: str
Alias for field number 1

year: str
Alias for field number 0

pudl.extract.epacems.**IGNORE_COLS = {'CO2_RATE', 'CO2_RATE (tons/mmBtu)', 'CO2_RATE_MEASURE_**
The set of EPA CEMS columns to ignore when reading data.

Type set

pudl.extract.epacems.**RENAME_DICT = {'CO2_MASS': 'co2_mass_tons', 'CO2_MASS (tons)': 'co2_m**
A dictionary containing EPA CEMS column names (keys) and replacement names to use when reading those columns into PUDL (values).

Type dict

pudl.extract.epacems.**extract**(*epacems_years*, *states*, *ds: pudl.workspace.datastore.Datastore*)
Coordinate the extraction of EPA CEMS hourly DataFrames.

Parameters

- **epacems_years** (*list*) – The years of CEMS data to extract, as 4-digit integers.
- **states** (*list*) – The states whose CEMS data we want to extract, indicated by 2-letter US state codes.
- **ds** (Datastore) – Initialized datastore

Yields *dict* – a dictionary with a single EPA CEMS tabular data resource name as the key, having the form "hourly_emissions_epacems_YEAR_STATE" where YEAR is a 4 digit number and STATE is a lower case 2-letter code for a US state. The value is a pandas.DataFrame containing all the raw EPA CEMS hourly emissions data for the indicated state and year.

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.

This module was written by @gschivley

class pudl.extract.epaipm.**EpaIpmDatastore**(*datastore: pudl.workspace.datastore.Datastore*)
Bases: object

Helper for extracting EpaIpm dataframes from Datastore.

SETTINGS = (TableSettings(table_name='transmission_single_epaipm', file='table_3-21_an

get_dataframe(*table_name: str*) → pandas.core.frame.DataFrame
Retrieve the specified file from the epaipm archive.

Parameters

- **table_name** – table name, from self.table_filename
- **pandas_args** – pandas arguments for parsing the file

Returns Pandas dataframe of EPA IPM data.

get_table_settings(*table_name: str*) → *pudl.extract.epaipm.TableSettings*
Returns TableSettings for a given table_name.

class pudl.extract.epaipm.**TableSettings**(*table_name: str, file: str, excel_settings: Dict[str, Any] = {}*)

Bases: `tuple`

Contains information for how to access and load EpaIpm dataframes.

excel_settings: Dict[str, Any]
Alias for field number 2

file: str
Alias for field number 1

table_name: str
Alias for field number 0

pudl.extract.epaipm.**extract**(*epaipm_tables: List[str], ds: pudl.workspace.datastore.Datastore*) → Dict[str, pandas.core.frame.DataFrame]

Extracts data from IPM files.

Parameters

- **epaipm_tables** (*iterable*) – A tuple or list of table names to extract
- **ds** (*EpaIpmDatastore*) – Initialized datastore

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

Return type dict

pudl.extract.excel module

Load excel metadata CSV files form a python data package.

class pudl.extract.excel.**GenericExtractor**(*ds*)

Bases: `object`

Contains logic for extracting panda.DataFrames from excel spreadsheets.

This class implements the generic dataset agnostic logic to load data from excel spreadsheet simply by using excel Metadata for given dataset.

It is expected that individual datasets wil subclass this code and add custom business logic by overriding necessary methods.

When implementing custom business logic, the following should be modified:

1. DATASET class attribute controls which excel metadata should be loaded.

2. BLACKLISTED_PAGES class attribute specifies which pages should not be loaded from the underlying excel files even if the metadata is available. This can be used for experimental/new code that should not be run yet.

3. dtypes() should return dict with {column_name: pandas_datatype} if you need to specify which datatypes should be uded upon loading.

4. If data cleanup is necessary, you can apply custom logic by overriding one of the following functions (they all return the modified dataframe):

- process_raw() is applied right after loading the excel DataFrame from the disk.
- process_renamed() is applied after input columns were renamed to standardized pudl columns.
- process_final_page() is applied when data from all available years is merged into single DataFrame for a given page.

5. get_datapackage_resources() if partition is anything other than a year, this method should be overwritten in the dataset-specific extractor.

BLACKLISTED_PAGES = []
List of supported pages that should not be extracted.

METADATA = None
Instance of metadata object to use with this extractor.

excel_filename(*page*, ***partition*)
Produce the xlsx document file name as it will appear in the archive.

Parameters

- **page** – pudl name for the dataset contents, eg "boiler_generator_assn" or "coal_stocks"
- **partition** – partition to load. (ex: 2009 for year partition or "2020-08" for year_month partition)

Returns string name of the xlsx file

extract(***partitions*)
Extracts dataframes.

Returns dict where keys are page names and values are DataFrames containing data across given years.

Parameters partitions (*list, tuple or string*) – list of partitions to extract. (Ex: [2009, 2010] if dataset is partitioned by years or '2020-08' if dataset is partitioned by year_month)

static get_dtypes(*page*, ***partition*)
Provide custom dtypes for given page and partition.

load_excel_file(*page*, ***partition*)
Produce the ExcelFile object for the given (partition, page).

Parameters

- **page** (*str*) – pudl name for the dataset contents, eg "boiler_generator_assn" or "coal_stocks"
- **partition** – partition to load. (ex: 2009 for year partition or "2020-08" for year_month partition)

Returns pd.ExcelFile instance with the parsed excel spreadsheet frame

static process_final_page(*df*, *page*)
Final processing stage applied to a page DataFrame.

process_raw(*df*, *page*, ***partition*)
Transforms raw dataframe and rename columns.

static process_renamed(*df*, *page*, ***partition*)
Transforms dataframe after columns are renamed.

class pudl.extract.excel.**Metadata**(*dataset_name*)
Bases: object

Load Excel metadata from Python package data.

Excel sheet files may contain many different tables. When we load those into dataframes, metadata tells us how to do this. Metadata generally informs us about the position of a given page in the file (which sheet and which row) and it informs us how to translate excel column names into standardized column names.

When metadata object is instantiated, it is given ${dataset} name and it will attempt to load csv files from pudl.package_data.meta.xlsx_maps.${dataset} package.

It expects the following kinds of files:

- skiprows.csv tells us how many initial rows should be skipped when loading data for given (partition, page).
- skipfooter.csv tells us how many bottom rows should be skipped when loading data for given partition (partition, page).
- tab_map.csv tells us what is the excel sheet name that should be read when loading data for given (partition, page)
- column_map/${page}.csv currently informs us how to translate input column names to standardized pudl names for given (partition, input_col_name). Relevant page is encoded in the filename.

get_all_columns(*page*)
Returns list of all pudl (standardized) columns for a given page (across all partition).

get_all_pages()
Returns list of all known pages.

get_column_map(*page*, ***partition*)
Returns the dictionary mapping input columns to pudl columns for given partition and page.

get_dataset_name()
Returns the name of the dataset described by this metadata.

get_sheet_name(*page*, ***partition*)
Returns name of the excel sheet that contains the data for given partition and page.

get_skipfooter(*page*, ***partition*)
Returns number of bottom rows to skip when loading given partition and page.

get_skiprows(*page*, ***partition*)
Returns number of initial rows to skip when loading given partition and page.

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 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.

Ths SQLite based compilation of the original FERC Form 1 databases can accommodate all 116 tables from all the published years of data (beginning in 1994). Including all the data through 2018, the database takes up more than 7GB of disk space. However, almost 90% of that "data" is embeded 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.

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*) –
- **()** –
- **()** –

class `pudl.extract.ferc1.`**Ferc1Datastore**(*datastore: pudl.workspace.datastore.Datastore*)

Bases: `object`

Simple datastore wrapper for accessing ferc1 resources.

PACKAGE_PATH = 'pudl.package_data.meta.ferc1_row_maps'

get_dir(*year: int*) → pathlib.Path

Returns the path where individual ferc1 files are stored inside the yearly archive.

get_file(*year: int*, *filename: str*)

Opens given ferc1 file from the corresponding archive.

`pudl.extract.ferc1.`**PUDL_RIDS = {514: 'AEP Texas', 519: 'Upper Michigan Energy Resources C**

Missing FERC 1 Respondent IDs for which we have identified the respondent.

`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*, *ds*, *refyear=2019*, *testing=False*, *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** (*sqlalchemy.schema.MetaData*) – The database schema to which the newly defined `sqlalchemy.Table` will be added.
- **dbc_map** (*dict*) – A dictionary of dictionaries
- **ds** (*Ferc1Datastore*) – Initialized datastore
- **testing** (*bool*) – Assume this is a test run, use sandboxes
- **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 None

`pudl.extract.ferc1.`**`check_ferc1_tables`**(*refyear*)

Test each FERC 1 data year for compatibility with reference year schema.

Parameters **refyear** (*int*) – The reference year for testing compatibility of the database schema with a FERC Form 1 table and year.

Returns A dictionary having database table names as keys, and lists of which years that table was compatible with the reference year as values.

Return type dict

`pudl.extract.ferc1.`**`dbf2sqlite`**(*tables*, *years*, *refyear*, *pudl_settings*, *bad_cols=()*, *clobber=False*, *datastore=None*)

Clone the FERC Form 1 Databsae 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.

- **datastore** (`Datastore`) – instance of a datastore to access the resources.

Returns None

`pudl.extract.ferc1.`**`define_sqlite_db`**(*sqlite_meta, dbc_map, ds, 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_fclty': '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_attsttn': '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_dfrrd_dr': 'F1_38', 'f1_miscgen_expnelc': 'F1_37', 'f1_mthly_peak_otpt': 'F1_39', 'f1_mtrl_spply': '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_dfrrd_cr': '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_r_d_demo_actvty': 'F1_59', 'f1_reconrpt_netinc': 'F1_55', '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': 'F1_RG_TRN_SRV_REV', 'f1_row_lit_tbl': 'F1_84', 'f1_s0_checks': 'F1_S0_CHECKS', 'f1_s0_filing_log':*

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.
- **ds** (*Ferc1Datastore*) – Initialized Ferc1Datastore
- **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** (*sqlalchemy.engine.Engine*) – A DB Engine pointing at an exising 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', 'purchased_power_ferc1', 'plant_in_service_ferc1'), ferc1_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, 2019), 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

`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`**(*ds*, *year*, *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_lengh`. 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.)

Parameters

- **`ds`** (*`Ferc1Datastore`*) – Initialized datastore
- **`year`** – 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.

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_ferc1_meta`**(*ferc1_engine*)

Grab the FERC Form 1 DB metadata and check that tables exist.

Connects to the FERC Form 1 SQLite database and reads in its metadata (table schemas, types, etc.) by reflecting the database. Checks to make sure the DB is not empty, and returns the metadata object.

Parameters **`ferc1_engine`** (*`sqlalchemy.engine.Engine`*) – SQL Alchemy database connection engine for the PUDL FERC 1 DB.

Returns sqlalchemy.Metadata A SQL Alchemy metadata object, containing the definition of the DB structure.

Raises `ValueError` – If there are no tables in the SQLite Database.

`pudl.extract.ferc1.get_fields(filedata)`
Produce the expected table names and fields from a DBC file.

Parameters **filedata** – Contents of the DBC file from which to extract.

Returns [fields]

Return type dict of table_name

`pudl.extract.ferc1.get_raw_df(ds, 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, 2019))`
Combine several years of a given FERC Form 1 DBF table into a dataframe.

Parameters

- **ds** (*Ferc1Datastore*) – Initialized datastore
- **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.
- **min_length** (*int*) – The minimum number of consecutive printable
- **years** (*list*) – 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.missing_respondents(reported, observed, identified)`
Fill in missing respondents for the f1_respondent_id table.

Parameters

- **reported** (*iterable*) – Respondent IDs appearing in f1_respondent_id.
- **observed** (*iterable*) – Respondent IDs appearing anywhere in the ferc1 DB.
- **identified** (*dict*) – A {respondent_id: respondent_name} mapping for those observed but not reported respondent IDs which we have been able to identify based on circumstantial evidence. See also: *pudl.extract.ferc1.PUDL_RIDS*

Returns A list of dictionaries representing minimal f1_respondent_id table records, of the form {"respondent_id": ID, "respondent_name": NAME}. These records are generated only for unreported respondents. Identified respondents get the values passed in through `identified` and the other observed but unidentified respondents are named "Missing Respondent ID"

Return type list

`pudl.extract.ferc1.observed_respondents(ferc1_engine)`
Compile the set of all observed respondent IDs found in the FERC 1 database.

A significant number of FERC 1 respondent IDs appear in the data tables, but not in the f1_respondent_id table. In order to construct a self-consisten database with we need to find all of those missing respondent IDs and inject them into the table when we clone the database.

Parameters **ferc1_engine** (*sqlalchemy.engine.Engine*) – An engine for connecting to the FERC 1 database.

Returns Every respondent ID reported in any of the FERC 1 DB tables.

Return type set

`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*)

Create a `pandas.DataFrame` containing valid raw f1_steam records.

Selected records must indicate a plant capacity greater than 0, and include a non-null plant name.

Parameters

- **`ferc1_meta`** (`sqlalchemy.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*, *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, 2019)*, *pk=('respondent_id', 'report_year', 'report_prd', 'row_number', 'spplmnt_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 spplmnt_num.

Returns None

pudl.extract.ferc714 module

Routines used for extracting the raw FERC 714 data.

`pudl.extract.ferc714.`**`TABLE_ENCODING = {'adjacency_ba_ferc714': 'iso-8859-1', 'demand_fore`**
Dictionary describing the character encodings of the FERC 714 CSV files.

`pudl.extract.ferc714.`**`TABLE_FNAME = {'adjacency_ba_ferc714': 'Part 2 Schedule 4 - Adjacent`**
Dictionary mapping PUDL tables to filenames within the FERC 714 zipfile.

`pudl.extract.ferc714.`**`extract`**(*tables=('respondent_id_ferc714', 'id_certification_ferc714', 'gen_plants_ba_ferc714', 'demand_monthly_ba_ferc714', 'net_energy_load_ba_ferc714', 'adjacency_ba_ferc714', 'interchange_ba_ferc714', 'lambda_hourly_ba_ferc714', 'lambda_description_ferc714', 'description_pa_ferc714', 'demand_forecast_pa_ferc714', 'demand_hourly_pa_ferc714'), pudl_settings=None, ds=None*)
Extract the raw FERC Form 714 dataframes from their original CSV files.

Parameters

- **`ferc714_tables`** (*`iterable`*) – The set of tables to be extracted.
- **`pudl_settings`** (*`dict`*) – A PUDL settings dictionary.
- **`ds`** (`Datastore`) – instance of the datastore

Returns A dictionary of dataframes, with raw FERC 714 table names as the keys, and minimally processed pandas.DataFrame instances as the values.

Return type dict

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 Sources*. 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.eia_epacems module

Extract, clean, and normalize the EPA-EIA crosswalk.

This module defines functions that read the raw EPA-EIA crosswalk file, clean up the column names, and separate it into three distinctive normalize tables for integration in the database. There are many gaps in the mapping of EIA plant and generator ids to EPA plant and unit ids, so, for the time being these tables are sparse.

The EPA, in conjunction with the EIA, plans to relase an crosswalk with fewer gaps at the beginning of 2021. Until then, this module reads and cleans the currently available crosswalk.

The raw crosswalk file was obtained from Greg Schivley. His methods for filling in some of the gaps are not included in this version of the module. https://github.com/grgmiller/EPA-EIA-Unit-Crosswalk

`pudl.glue.eia_epacems.`**`grab_clean_split`**`()`

Clean raw crosswalk data, drop nans, and return split tables.

Returns a dictionary of three normalized DataFrames comprised of the data in the original crosswalk file. EPA plant id to EPA unit id; EPA plant id to EIA plant id; and EIA plant id to EIA generator id to EPA unit id.

Return type dict

`pudl.glue.eia_epacems.`**`grab_n_clean_epa_orignal`**`()`

Retrieve and clean column names for the original EPA-EIA crosswalk file.

Returns

a version of the EPA-EIA crosswalk containing only relevant columns. Columns names are clear and programatically accessible.

Return type pandas.DataFrame

`pudl.glue.eia_epacems.`**`split_tables`**`(`*df*`)`

Split the cleaned EIA-EPA crosswalk table into three normalized tables.

Parameters **`pandas.DataFrame`** – a DataFrame of relevant, readible columns from the EIA-EPA crosswalk. Output of grab_n_clean_epa_original().

Returns a dictionary of three normalized DataFrames comprised of the data in the original crosswalk file. EPA plant id to EPA unit id; EPA plant id to EIA plant id; and EIA plant id to EIA generator id to EPA unit id. Includes no nan values.

Return type dict

pudl.glue.ferc1_eia module

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

FERC1 and EIA report on many of 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 thousands of reported plants, we know there are probably some incorrect or incomplete mappings. If you see a *plant_id_pudl* or *utility_id_pudl* mapping that you think is incorrect, please open an issue on our Github!

Note that the PUDL IDs may change over time. They are not guaranteed to be stable. If you need to find a particular plant or utility reliably, you should use its plant_id_eia, utility_id_eia, or utility_id_ferc1.

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.`**`get_db_plants_eia`**`(`*pudl_engine*`)`

Get a list of all EIA plants appearing in the PUDL DB.

This list of plants is used to determine which plants need to be added to the FERC 1 / EIA plant mappings, where we assign PUDL Plant IDs. Unless a new year's worth of data has been added to the PUDL DB, but the plants have not yet been mapped, all plants in the PUDL DB should also appear in the plant mappings. It only makes sense to run this with a connection to a PUDL DB that has all the EIA data in it.

Parameters **pudl_engine** (*sqlalchemy.engine.Engine*) – A database connection engine for connecting to a PUDL SQLite database.

Returns A DataFrame with plant_id_eia, plant_name_eia, and state columns, for addition to the FERC 1 / EIA plant mappings.

Return type pandas.DataFrame

`pudl.glue.ferc1_eia.`**`get_db_plants_ferc1`**`(`*pudl_settings*, *years*`)`

Pull a dataframe of all plants in the FERC Form 1 DB for the given years.

This function looks in the f1_steam, f1_gnrt_plant, f1_hydro and f1_pumped_storage tables, and generates a dataframe containing every unique combination of respondent_id (utility_id_ferc1) and plant_name is finds. Also included is the capacity of the plant in MW (as reported in the raw FERC Form 1 DB), the respondent_name (utility_name_ferc1) and a column indicating which of the plant tables the record came from. Plant and utility names are translated to lowercase, with leading and trailing whitespace stripped and repeating internal whitespace compacted to a single space.

This function is primarily meant for use generating inputs into the manual mapping of FERC to EIA plants with PUDL IDs.

Parameters

- **pudl_settings** (*dict*) – Dictionary containing various paths and database URLs used by PUDL.
- **years** (*iterable*) – Years for which plants should be compiled.

Returns A dataframe containing columns utility_id_ferc1, utility_name_ferc1, plant_name, capacity_mw, and plant_table. Each row is a unique combination of utility_id_ferc1 and plant_name.

Return type pandas.DataFrame

`pudl.glue.ferc1_eia.`**`get_db_utils_eia`**`(`*pudl_engine*`)`

Get a list of all EIA Utilities appearing in the PUDL DB.

`pudl.glue.ferc1_eia.`**`get_lost_plants_eia`**`(`*pudl_engine*`)`

Identify any EIA plants which were mapped, but then lost from the DB.

`pudl.glue.ferc1_eia.`**`get_lost_utils_eia`**`(`*pudl_engine*`)`

Get a list of all mapped EIA Utilites not found in the PUDL DB.

`pudl.glue.ferc1_eia.`**`get_mapped_plants_eia`**`()`

Get a list of all EIA plants that have been assigned PUDL Plant IDs.

Read in the list of already mapped EIA plants from the FERC 1 / EIA plant and utility mapping spreadsheet kept in the package_data.

Parameters **None** –

Returns A DataFrame listing the plant_id_eia and plant_name_eia values for every EIA plant which has already been assigned a PUDL Plant ID.

Return type pandas.DataFrame

`pudl.glue.ferc1_eia.`**`get_mapped_plants_ferc1`**`()`

Generate a dataframe containing all previously mapped FERC 1 plants.

Many plants are reported in FERC Form 1 with different versions of the same name in different years. Because FERC provides no unique ID for plants, these names must be used as part of their identifier. We manually curate a list of all the versions of plant names which map to the same actual plant. In order to identify new plants each year, we have to compare the new plant names and respondent IDs against this raw mapping, not the contents of the PUDL data, since within PUDL we use one canonical name for the plant. This function pulls that list of various plant names and their corresponding utilities (both name and ID) for use in identifying which plants have yet to be mapped when we are integrating new data.

Parameters **None** –

Returns plant_name, utility_id_ferc1, and utility_name_ferc1. Each row represents a unique combination of utility_id_ferc1 and plant_name.

Return type pandas.DataFrame A DataFrame with three columns

`pudl.glue.ferc1_eia.`**`get_mapped_utils_eia`**`()`

Get a list of all the EIA Utilities that have PUDL IDs.

`pudl.glue.ferc1_eia.`**`get_mapped_utils_ferc1`**`()`

Read in the list of manually mapped utilities for FERC Form 1.

Unless a new utility has appeared in the database, this should be identical to the full list of utilities available in the FERC Form 1 database.

Parameters **None** –

Returns pandas.DataFrame

`pudl.glue.ferc1_eia.`**`get_plant_map`**`()`

Read in the manual FERC to EIA plant mapping data.

`pudl.glue.ferc1_eia.`**`get_unmapped_plants_eia`**`(`*pudl_engine*`)`

Identify any as-of-yet unmapped EIA Plants.

`pudl.glue.ferc1_eia.`**`get_unmapped_plants_ferc1`**`(`*pudl_settings*, *years*`)`

Generate a DataFrame of all unmapped FERC plants in the given years.

Pulls all plants from the FERC Form 1 DB for the given years, and compares that list against the already mapped plants. Any plants found in the database but not in the list of mapped plants are returned.

Parameters

- **pudl_settings** (*dict*) – Dictionary containing various paths and database URLs used by PUDL.
- **years** (*iterable*) – Years for which plants should be compiled from the raw FERC Form 1 DB.

Returns A dataframe containing five columns: utility_id_ferc1, utility_name_ferc1, plant_name, capacity_mw, and plant_table. Each row is a unique combination of utility_id_ferc1 and plant_name, which appears in the FERC Form 1 DB, but not in the list of manually mapped plants.

Return type pandas.DataFrame

`pudl.glue.ferc1_eia.`**`get_unmapped_utils_eia`**`(`*pudl_engine*`)`

Get a list of all the EIA Utilities in the PUDL DB without PUDL IDs.

`pudl.glue.ferc1_eia.`**`get_unmapped_utils_ferc1`**`(`*ferc1_engine*`)`

Generate a list of as-of-yet unmapped utilities from the FERC Form 1 DB.

Find any utilities which do exist in the cloned FERC Form 1 DB, but which do not show up in the already mapped FERC respondents.

Parameters **ferc1_engine** (*sqlalchemy.engine.Engine*) – A database connection engine for the cloned FERC Form 1 DB.

Returns with columns "utility_id_ferc1" and "utility_name_ferc1"

Return type pandas.DataFrame

`pudl.glue.ferc1_eia.`**`get_unmapped_utils_with_plants_eia`**(*pudl_engine*)

Get all EIA Utilities that lack PUDL IDs but have plants/ownership.

`pudl.glue.ferc1_eia.`**`get_utility_map`**()

Read in the manual FERC to EIA utility mapping data.

`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

Functions for loading processed PUDL data tables into CSV files.

Once each set of tables pertaining to a data source have been transformed, we need to output them into CSV files which will become the data underlying tabular data resources. Most of these resources contain an entire table. In the case of larger tables (like EPA CEMS) the data may be partitioned into a collection of gzipped CSV files which are all part of a single resource group.

These functions are designed to pick up where the transform step leaves off, taking a dictionary of dataframes and applying a few last alterations that are necessary only in the context of outputting the data as text based files. These include converting floatified integer columns into strings with null values, and appropriately indexing the dataframes as needed.

`pudl.load.csv.`**`clean_columns_dump`**(*df*, *resource_name*, *datapkg_dir*)

Output cleaned data columns to a CSV file.

Ensures that the id column is set appropriately depending on whether the table has a natural primary key or an autoincremnted pseudo-key. Ensures that the set of columns in the dataframe to be output are identical to those in the corresponding metadata definition. Transforms integer columns with NA values into strings for dumping, as appropriate.

Parameters

- **resource_name** (*str*) – The exact name of the tabular resource which the DataFrame df is going to be used to populate. This will be used to name the output CSV file, and must match the corresponding stored metadata template.
- **datapkg_dir** (*path-like*) – Path to the datapackage directory that the CSV will be part of. Assumes CSV files get put in a "data" directory within this directory.
- **df** (*pandas.DataFrame*) – The dataframe containing the data to be written out into CSV for inclusion in a tabular datapackage.

Returns None

`pudl.load.csv.`**`csv_dump`**(*df*, *resource_name*, *keep_index*, *datapkg_dir*)

Write a dataframe to CSV.

Set `pandas.DataFrame.to_csv()` arguments appropriately depending on what data source we're writing out, and then write it out. In practice this means adding a .csv to the end of the resource name, and then, if it's part of epacems, adding a .gz after that.

Parameters

- **df** (*pandas.DataFrame*) – The DataFrame to be dumped to CSV.
- **resource_name** (*str*) – The exact name of the tabular resource which the DataFrame df is going to be used to populate. This will be used to name the output CSV file, and must match the corresponding stored metadata template.

- **keep_index** (*bool*) – if True, use the "id" column of df as the index and output it.
- **datapkg_dir** (*path-like*) – Path to the top level datapackage directory.

Returns None

`pudl.load.csv.`**`dict_dump`**(*transformed_dfs*, *data_source*, *datapkg_dir*)

Wrapper for clean_columns_dump that takes a dictionary of DataFrames.

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)
- **data_source** (*str*) – The name of the data source we are working with (eia923, ferc1, etc.)
- **datapkg_dir** (*path-like*) – Path to the top level directory for the datapackage these CSV files are part of. Will contain a "data" directory and a datapackage.json file.

Returns None

pudl.load.metadata module

Routines for generating PUDL tabular data package and resource metadata.

This module enables the generation and use of the metadata for tabular data packages. It also saves and validates the datapackage once the metadata is compiled. In general the routines in this module can only be used **after** the referenced CSV's have been generated by the top level PUDL ETL module, and written out to the datapackage data directory by the *pudl.load.csv* module.

The metadata comes from three basic sources: the datapkg_settings that are read in from the YAML file specifying the datapackage or bundle of datapackages to be generated, the CSV files themselves (their names, sizes, and hash values) and the stored metadata template which ultimately determines the structure of the relational database that these output tabular data packages represent, and encodes field specific table schemas. See the "megadata" which is stored in *src/pudl/package_data/meta/datapkg/datapackage.json*.

For unpartitioned tables which are contained in a single tabular data resource this is a relatively straightforward process. However, larger tables that have been partitioned into smaller tabular data resources that are part of a resource group (e.g. EPA CEMS) have additional complexities. We have tried to say "resource" when referring to an individual output CSV that has its own metadata entry, and "table" when referring to whole tables which typically contain only a single resource, but may be composed of hundreds or even thousands of individual resources.

See https://frictionlessdata.io for more details on the tabular data package standards.

In addition, we have included PUDL specific metadata fields that document the ETL parameters which were used to process the data, temporal and spatial coverage for each resource, Zenodo DOIs if appropriate, UUIDs to identify the individual data packages as well as co-generated bundles of data packages that can be used together to instantiate a single database, etc.

`pudl.load.metadata.`**`compile_keywords`**(*data_sources*)

Compile the set of all keywords associated with given data sources.

The list of keywords we associate with each data source is stored in the `pudl.constants.keywords_by_data_source` dictionary.

Parameters data_sources (*iterable*) – List of data source codes (eia923, ferc1, etc.) from which to gather keywords.

Returns the set of all unique keywords associated with any of the input data sources.

Return type list

`pudl.load.metadata.`**`compile_partitions`**(*datapkg_settings*)

Given a datapackage settings dictionary, extract dataset partitions.

Iterates through all the datasets enumerated in the datapackage settings, and compiles a dictionary indicating which datasets should be partitioned and on what basis when they are output as tabular data resources. Currently this only applies to the epacems dataset. Datapackage settings must be validated because currently we inject EPA CEMS partitioning variables (epacems_years, epacems_states) during the validation process.

Parameters **`datapkg_settings`** (*`dict`*) – a dictionary containing validated datapackage settings, mostly read in from a PUDL ETL settings file.

Returns Uses table name (e.g. hourly_emissions_epacems) as keys, and lists of partition variables (e.g. ["epacems_years", "epacems_states"]) as the values. If no datasets within the datapackage are being partitioned, this is an empty dictionary.

Return type dict

`pudl.load.metadata.`**`data_sources_from_tables`**(*table_names*)

Look up data sources used by the given list of PUDL database tables.

Parameters **`tables_names`** (*`iterable`*) – a list of names of 'seed' tables, whose dependencies we are seeking to find.

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

Return type set

`pudl.load.metadata.`**`generate_metadata`**(*datapkg_settings*, *datapkg_resources*, *datapkg_dir*, *datapkg_bundle_uuid=None*, *datapkg_bundle_doi=None*)

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

- **`datapkg_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, unique package name e.g. pudl-eia923, ferc1-test * title: One line human readable description. * description: A paragraph long description. * version: the version of the data package being published. * keywords: For search purposes.
- **`datapkg_resources`** (*`list`*) – The names of tabular data resources that are included in this data package.
- **`datapkg_dir`** (*`path-like`*) – The location of the directory for this package. The data package directory will be a subdirectory in the *datapkg_dir* directory, with the name of the package as the name of the subdirectory.
- **`datapkg_bundle_uuid`** – A type 4 UUID identifying the ETL run which which generated the data package – this indicates that the data packages are compatible with each other
- **`datapkg_bundle_doi`** – A digital object identifier (DOI) that will be used to archive the bundle of mutually compatible data packages. Needs to be provided by an archiving service like Zenodo. This field may also be added after the data package has been generated.

Returns a Python dictionary representing a valid tabular data package descriptor.

Return type dict

`pudl.load.metadata.`**`get_autoincrement_columns`**(*unpartitioned_tables*)
Grab the autoincrement columns for pkg tables.

`pudl.load.metadata.`**`get_datapkg_fks`**(*datapkg_json*)
Get a dictionary of foreign key relationships from datapackage metadata.

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

Returns

table names (keys) with lists of table names (values) which the key table has forgien key relationships with.

Return type dict

`pudl.load.metadata.`**`get_dependent_tables`**(*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`** (*`dict`*) – table names (keys) with lists of table names (values) which the key table has forgien key relationships with.

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

Return type set

`pudl.load.metadata.`**`get_dependent_tables_from_list`**(*table_names*)
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.

Returns All tables with which any of the input tables have ForeignKey relations.

Return type set

`pudl.load.metadata.`**`get_tabular_data_resource`**(*resource_name*, *datapkg_dir*, *datapkg_settings*, *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

- **`resource_name`** (*`string`*) – name of the tabular data resource for which you want to generate a Tabular Data Resource descriptor. This is the resource name, rather than the database table name, because we partition large tables into resource groups consisting of many files.
- **`datapkg_dir`** (*`path-like`*) – The location of the directory for this package. The data package directory will be a subdirectory in the *datapkg_dir* directory, with the name of the package as the name of the subdirectory.

- **datapkg_settings** (*dict*) – Python dictionary represeting the ETL parameters read in from the settings file, pertaining to the tabular datapackage this resource is part of.
- **partitions** (*dict*) – A dictionary with PUDL database table names as the keys (e.g. hourly_emissions_epacems), and lists of partition variables (e.g. ["epacems_years", "epacems_states"]) as the keys.

Returns A Python dictionary representing a tabular data resource descriptor that complies with the Frictionless Data specification.

Return type dict

pudl.load.metadata.**get_unpartitioned_tables**(*resources*, *datapkg_settings*)

Generate a list of database table names from a list of data resources.

In the case of EPA CEMS and potentially other large datasets, we are partitioning a single table into many tabular data resources that are part of a resource group. However in some contexts we want to refer to the list of corresponding databse tables, rather than the list of resources.

The partition key in the datapackage settings is the name of the table without the partition elements, and so in the case of partitioned tables we use that key as the name of the table. Otherwise we just use the name of the resource.

Parameters

- **resources** (*iterable*) – A list of tabular data resource names. They must be expected to appear in the datapackage specified by datapkg_settings.
- **datapkg_settings** (*dict*) – a dictionary containing validated datapackage settings, mostly read in from a PUDL ETL settings file.

Returns

The names of the database tables corresponding to the tabular datapackage resource names that were passed in.

Return type list

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.**pull_resource_from_megadata**(*resource_name*)

Read metadata for a given data resource from the stored PUDL megadata.

Parameters **resource_name** (*str*) – the name of the tabular data resource whose JSON descriptor we are reading.

Returns A Python dictionary containing the resource descriptor portion of a data package descriptor, not expected to be valid or complete.

Return type dict

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

pudl.load.metadata.**spatial_coverage**(*resource_name*)

Extract spatial coverage (country and state) for a given source.

Parameters **resource_name** (*str*) – The name of the (potentially partitioned) resource for which we are enumerating the spatial coverage. Currently this is the only place we are able to access the partitioned spatial coverage after the ETL process has completed.

Returns A dictionary containing country and potentially state level spatial coverage elements. Country keys are "country" for the full name of country, "iso_3166-1_alpha-2" for the 2-letter ISO code, and "iso_3166-1_alpha-3" for the 3-letter ISO code. State level elements are "state" (a two letter ISO code for sub-national jurisdiction) and "iso_3166-2" for the combined country-state code conforming to that standard.

Return type dict

`pudl.load.metadata.`**`temporal_coverage`**(*resource_name*, *datapkg_settings*)

Extract start and end dates from ETL parameters for a given source.

Parameters

- **resource_name** (*str*) – The name of the (potentially partitioned) resource for which we are enumerating the spatial coverage. Currently this is the only place we are able to access the partitioned spatial coverage after the ETL process has completed.
- **datapkg_settings** (*dict*) – Python dictionary represeting the ETL parameters read in from the settings file, pertaining to the tabular datapackage this resource is part of.

Returns A dictionary of two items, keys "start_date" and "end_date" with values in ISO 8601 YYYY-MM-DD format, indicating the extent of the time series data contained within the resource. If the resource does not contain time series data, the dates are null.

Return type dict

`pudl.load.metadata.`**`validate_save_datapkg`**(*datapkg_descriptor*, *datapkg_dir*)

Validate datapackage descriptor, save it, and validate some sample data.

Parameters

- **datapkg_descriptor** (*dict*) – A Python dictionary representation of a (hopefully valid) tabular datapackage descriptor.
- **datapkg_dir** (*path-like*) – Directory into which the datapackage.json file containing the tabular datapackage descriptor should be written.

Returns A dictionary containing the goodtables datapackage validation report. Note that this will only be returned if there are no errors, otherwise it is output as an error message.

Return type dict

Raises **ValueError** – if the datapackage descriptor passed in is invalid, or if any of the tables has a data validation error.

Module contents

Tools for handling the load set in pudl ETL.

pudl.output package

Submodules

pudl.output.censusdp1tract module

Functions for reading data out of the Census DP1 SQLite Database.

`pudl.output.censusdp1tract.`**`get_layer`**(*layer: Literal[state, county, tract]*, *pudl_settings=None*) → geopandas.geodataframe.GeoDataFrame

Select one layer from the Census DP1 database.

Uses information within the Census DP1 database to set the coordinate reference system and to identify the column containing the geometry. The geometry column is renamed to "geom" as that's the default withing Geopandas. No other column names or types are altered.

Parameters

- **`layer`** (*`str`*) – Which set of geometries to read, must be one of "state", "county", or "tract".
- **`pudl_settings`** (*`dict or None`*) – A dictionary of PUDL settings, including paths to various resources like the Census DP1 SQLite database. If None, the user defaults are used.

Returns geopandas.GeoDataFrame

pudl.output.eia860 module

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

`pudl.output.eia860.`**`boiler_generator_assn_eia860`**(*pudl_engine*, *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.
- **`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 the fields from the EIA 860 boiler generator association table.

Return type pandas.DataFrame

`pudl.output.eia860.`**`generators_eia860`**(*pudl_engine*, *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.
- **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 the fields of the EIA 860 Generators table.

Return type pandas.DataFrame

`pudl.output.eia860.`**`ownership_eia860`**(*pudl_engine*, *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.
- **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 a useful set of fields related to the EIA 860 Ownership table.

Return type pandas.DataFrame

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

Pull all fields from the EIA Plants tables.

Parameters

- **pudl_engine** (*sqlalchemy.engine.Engine*) – SQLAlchemy connection engine for the PUDL DB.
- **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 the fields of the EIA 860 Plants table.

Return type pandas.DataFrame

`pudl.output.eia860.`**`plants_utils_eia860`**(*pudl_engine*, *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_eia (from EIA entity) - plant_id_eia (from EIA entity) - plant_id_pudl - utility_id_eia (from EIA860) - utility_name_eia (from EIA860) - utility_id_pudl

Parameters

- **pudl_engine** (*sqlalchemy.engine.Engine*) – SQLAlchemy connection engine for the PUDL DB.
- **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 plant and utility IDs and names from EIA 860.

Return type pandas.DataFrame

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

Pull all fields from the EIA860 Utilities table.

Parameters

- **pudl_engine** (*sqlalchemy.engine.Engine*) – SQLAlchemy connection engine for the PUDL DB.
- **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 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.**FUEL_COST_CATEGORIES_EIAAPI = [41696, 41762, 41740]**

The category ids for fuel costs by fuel for electricity for coal, gas and oil.

Each category id is a peice of a query to EIA's API. Each query here contains a set of state-level child series which contain fuel cost data.

See EIA's query browse here:

- Coal: https://www.eia.gov/opendata/qb.php?category=41696
- Gas: https://www.eia.gov/opendata/qb.php?category=41762
- Oil: https://www.eia.gov/opendata/qb.php?category=41740

pudl.output.eia923.**boiler_fuel_eia923**(*pudl_engine*, *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.
- **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.`**`convert_cost_json_to_df`**(*response_fuel_state_annual*)

Convert a fuel-type/state response into a clean dataframe.

Parameters **response_fuel_state_annual** (*api response*) – an EIA API response which contains state-level series including monthly fuel cost data.

Returns a dataframe containing state-level montly fuel cost. The table contains the following columns, some of which are refernce columns: 'report_date', 'fuel_cost_per_unit', 'state', 'fuel_type_code_pudl', 'units' (ref), 'series_id' (ref), 'name' (ref).

Return type pandas.DataFrame

`pudl.output.eia923.`**`fuel_receipts_costs_eia923`**(*pudl_engine, freq=None, start_date=None, end_date=None, fill=False, roll=False*)

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.
- **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.
- **fill** (*boolean*) – if set to True, fill in missing coal, gas and oil fuel cost per mmbtu from EIA's API. This fills with montly state-level averages.
- **roll** (*boolean*) – if set to True, apply a rolling average to a subset of output table's columns (currently only 'fuel_cost_per_mmbtu' for the frc table).

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*, *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.
- **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, 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.
- **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.eia923.get_fuel_cost_avg_eiaapi(fuel_cost_cat_ids)`

Get a dataframe of state-level average fuel costs for EIA's API.

Parameters **fuel_cost_cat_ids** (*list*) – list of category ids. Known/testing working ids are stored in FUEL_COST_CATEGORIES_EIAAPI.

Returns a dataframe containing state-level montly fuel cost. The table contains the following columns, some of which are refernce columns: 'report_date', 'fuel_cost_per_unit', 'state', 'fuel_type_code_pudl', 'units' (ref), 'series_id' (ref), 'name' (ref).

Return type pandas.DataFrame

`pudl.output.eia923.get_response(url)`

Get a response from the API's url.

`pudl.output.eia923.`**`grab_fuel_state_monthly`**(*cat_id*)

Grab an API response for monthly fuel costs for one fuel category.

The data we want from EIA is in monthly, state-level series for each fuel type. For each fuel category, there are at least 51 embeded child series. This function compiles one fuel type's child categories into one request. The resulting api response should contain a list of series responses from each state which we can convert into a pandas.DataFrame using convert_cost_json_to_df.

Parameters **`cat_id`** (*`int`*) – category id for one fuel type. Known to be

`pudl.output.eia923.`**`make_url_cat_eiaapi`**(*category_id*)

Generate a url for a category from EIA's API.

`pudl.output.eia923.`**`make_url_series_eiaapi`**(*series_id*)

Generate a url for a series EIA's API.

pudl.output.epacems module

Routines that provide user-friendly access to the partitioned EPA CEMS dataset.

`pudl.output.epacems.`**`get_plant_states`**(*plant_ids*, *pudl_out*)

Determine what set of states a given set of EIA plant IDs are within.

If you only want to select data about a particular set of power plants from the EPA CEMS data, this is useful for identifying which patitions of the Parquet dataset you will need to search.

Parameters

- **`plant_ids`** (*`iterable`*) – A collection of integers representing valid plant_id_eia values within the PUDL DB.
- **`pudl_out`** (`pudl.output.pudltabl.PudlTabl`) – A PudlTabl output object to use to access the PUDL DB.

Returns A list containing the 2-letter state abbreviations for any state that was found in association with one or more of the plant_ids.

Return type list

`pudl.output.epacems.`**`get_plant_years`**(*plant_ids*, *pudl_out*)

Determine which years a given set of EIA plant IDs appear in.

If you only want to select data about a particular set of power plants from the EPA CEMS data, this is useful for identifying which patitions of the Parquet dataset you will need to search.

NOTE: the EIA-860 and EIA-923 data which are used here don't cover as many years as the EPA CEMS, so this is probably of limited utility – you may want to simply include all years, or manually specify the years of interest instead.

Parameters

- **`plant_ids`** (*`iterable`*) – A collection of integers representing valid plant_id_eia values within the PUDL DB.
- **`pudl_out`** (`pudl.output.pudltabl.PudlTabl`) – A PudlTabl output object to use to access the PUDL DB.

Returns A list containing the 4-digit integer years found in association with one or more of the plant_ids.

Return type list

pudl.output.epacems.**year_state_filter**(*years=()*, *states=()*)
Create filters to read given years and states from partitioned parquet dataset.

A subset of an Apache Parquet dataset can be read in more efficiently if files which don't need to be queried are avoideed. Some datasets are partitioned based on the values of columns to make this easier. The EPA CEMS dataset which we publish is partitioned by state and report year.

However, the way the filters are specified can be unintuitive. They use DNF (disjunctive normal form) See this blog post for more details:

https://blog.datasyndrome.com/python-and-parquet-performance-e71da65269ce

This function takes a set of years, and a set of states, and returns a list of lists of tuples, appropriate for use with the read_parquet() methods of pandas and dask dataframes. The filter will include all combinations of the specified years and states. E.g. if years=(2018, 2019) and states=("CA", "CO") then the filter would result in getting 2018 and 2019 data for CO, as well as 2018 and 2019 data for CA.

Parameters

- **years** (*iterable*) – 4-digit integers indicating the years of data you would like to read. By default it includes all years.
- **states** (*iterable*) – 2-letter state abbreviations indicating what states you would like to include. By default it includes all states.

Returns A list of lists of tuples, suitable for use as a filter in the read_parquet method of pandas and dask dataframes.

Return type list

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*, *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_ferc1) related to fuel consumption.

Parameters

- **pudl_engine** (*sqlalchemy.engine.Engine*) – Engine for connecting to the PUDL database.
- **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*)
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.

Useful derived values include:

- fuel_consumed_mmbtu (total fuel heat content consumed)

- `fuel_consumed_total_cost` (total cost of that fuel)

  Parameters **pudl_engine** (*sqlalchemy.engine.Engine*) – Engine for connecting to the PUDL database.

  Returns A DataFrame containing useful FERC Form 1 fuel information.

  Return type pandas.DataFrame

`pudl.output.ferc1.`**`plant_in_service_ferc1`**(*pudl_engine*)
Pull a dataframe of FERC Form 1 Electric Plant in Service data.

`pudl.output.ferc1.`**`plants_hydro_ferc1`**(*pudl_engine*)
Pull a useful dataframe related to the FERC Form 1 hydro plants.

`pudl.output.ferc1.`**`plants_pumped_storage_ferc1`**(*pudl_engine*)
Pull a dataframe of FERC Form 1 Pumped Storage plant data.

`pudl.output.ferc1.`**`plants_small_ferc1`**(*pudl_engine*)
Pull a useful dataframe related to the FERC Form 1 small plants.

`pudl.output.ferc1.`**`plants_steam_ferc1`**(*pudl_engine*)
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.

Also calculates `capacity_factor` (based on `net_generation_mwh` & `capacity_mw`)

Parameters **pudl_engine** (*sqlalchemy.engine.Engine*) – Engine for connecting to the PUDL database.

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*)
Build a dataframe of useful FERC Plant & Utility information.

Parameters **pudl_engine** (*sqlalchemy.engine.Engine*) – Engine for connecting to the PUDL database.

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

Return type pandas.DataFrame

`pudl.output.ferc1.`**`purchased_power_ferc1`**(*pudl_engine*)
Pull a useful dataframe of FERC Form 1 Purchased Power data.

pudl.output.ferc714 module

Functions & classes for compiling derived aspects of the FERC Form 714 data.

`pudl.output.ferc714.`**`ASSOCIATIONS: List[Dict[str, Any]] = [{'id': 56669, 'from': 2011, 'to`**
Adjustments to balancing authority-utility associations from EIA 861.

The changes are applied locally to EIA 861 tables.

- *id* (int): EIA balancing authority identifier (*balancing_authority_id_eia*).
- *from* (int): Reference year, to use as a template for target years.

- *to* (List[int]): Target years, in the closed interval format [minimum, maximum]. Rows in *balancing_authority_eia861* are added (if missing) for every target year with the attributes from the reference year. Rows in *balancing_authority_assn_eia861* are added (or replaced, if existing) for every target year with the utility associations from the reference year. Rows in *service_territory_eia861* are added (if missing) for every target year with the nearest year's associated utilities' counties.
- *exclude* (Optional[List[str]]): Utilities to exclude, by state (two-letter code). Rows are excluded from *balancing_authority_assn_eia861* with target year and state.

class pudl.output.ferc714.**Respondents**(*pudl_out*, *pudl_settings=None*, *ba_ids=None*, *util_ids=None*, *priority='balancing_authority'*, *limit_by_state=True*)

Bases: `object`

A class coordinating compilation of data related to FERC 714 Respondents.

The FERC 714 Respondents themselves are not complex as they are reported, but various ambiguities and the need to associate service territories with them mean there are a lot of different derived aspects related to them which we repeatedly need to compile in a self consistent way. This class allows you to choose several parameters for that compilation, and then easily access the resulting derived tabular outputs.

Some of these derived attributes are computationally expensive, and so they are cached internally. You can force a new computation in most cases by using `update=True` in the access methods. However, this functionality isn't totally implemented because we're still depending on the interim ETL processes for the FERC 714 and EIA 861 data, and we don't want to trigger whole new ETL runs every time a derived value is updated.

pudl_out

The PUDL output object which should be used to obtain PUDL data.

Type *pudl.output.pudltabl.PudlTabl*

pudl_settings

A dictionary of settings indicating where data related to PUDL can be found. Needed to obtain US Census DP1 data which has the county geometries.

Type dict or None

ba_ids

EIA IDs that should be treated as referring to balancing authorities in respondent categorization process. If None, all known values of `balancing_authority_id_eia` will be used.

Type ordered collection or None

util_ids

EIA IDs that should be treated as referring to utilities in respondent categorization process. If None, all known values of `utility_id_eia` will be used.

Type ordered collection or None

priority

Which type of entity should take priority in the categorization of FERC 714 respondents. Must be either `utility` or `balancing_authority`. The default is `balancing_authority`.

Type str

limit_by_state

Whether to limit respondent service territories to the states where they have documented activity in the EIA 861. Currently this is only implemented for Balancing Authorities.

Type bool

annualize(*update=False*)

Broadcast respondent data across all years with reported demand.

The FERC 714 Respondent IDs and names are reported in their own table, without any refence to individual years, but much of the information we are associating with them varies annually. This method creates an annualized version of the respondent table, with each respondent having an entry corresponding to every year in which hourly demand was reported in the FERC 714 dataset as a whole – this necessarily means that many of the respondents will end up having entries for years in which they reported no demand, and that's fine. They can be filtered later.

property balancing_authority_assn_eia861
Modified balancing_authority_assn_eia861 table.

property balancing_authority_eia861
Modified balancing_authority_eia861 table.

categorize(*update=False*)
Annualized respondents with `respondent_type` assigned if possible.

Categorize each respondent as either a `utility` or a `balancing_authority` using the parameters stored in the instance of the class. While categorization can also be done without annualizing, this function annualizes as well, since we are adding the `respondent_type` in order to be able to compile service territories for the respondent, which vary annually.

fipsify(*update=False*)
Annual respondents with the county FIPS IDs for their service territories.

Given the `respondent_type` associated with each respondent (either `utility` or `balancing_authority`) compile a list of counties that are part of their service territory on an annual basis, and merge those into the annualized respondent table. This results in a very long dataframe, since there are thousands of counties and many of them are served by more than one entity.

Currently respondents categorized as `utility` will include any county that appears in the `service_territory_eia861` table in association with that utility ID in each year, while for `balancing_authority` respondents, some counties can be excluded based on state (if `self.limit_by_state==True`).

georef_counties(*update=False*)
Annual respondents with all associated county-level geometries.

Given the county FIPS codes associated with each respondent in each year, pull in associated geometries from the US Census DP1 dataset, so we can do spatial analyses. This keeps each county record independent – so there will be many records for each respondent in each year. This is fast, and still good for mapping, and retains all of the FIPS IDs so you can also still do ID based analyses.

georef_respondents(*update=False*)
Annual respondents with a single all-encompassing geometry for each year.

Given the county FIPS codes associated with each responent in each year, compile a geometry for the respondent's entire service territory annually. This results in just a single record per respondent per year, but is computationally expensive and you lose the information about what all counties are associated with the respondent in that year. But it's useful for merging in other annual data like total demand, so you can see which respondent-years have both reported demand and decent geometries, calculate their areas to see if something changed from year to year, etc.

property service_territory_eia861
Modified service_territory_eia861 table.

summarize_demand(*update=False*)
Compile annualized, categorized respondents and summarize values.

Calculated summary values include: * Total reported electricity demand per respondent (`demand_annual_mwh`) * Reported per-capita electrcity demand

(demand_annual_per_capita_mwh) * Population density (population_density_km2) * Demand density (demand_density_mwh_km2)

These metrics are helpful identifying suspicious changes in the compiled annual geometries for the planning areas.

pudl.output.ferc714.**UTILITIES: List[Dict[str, Any]] = [{'id': 14328, 'reassign': True},**

Balancing authorities to treat as utilities in associations from EIA 861.

The changes are applied locally to EIA 861 tables.

- *id* (int): EIA balancing authority (BA) identifier (*balancing_authority_id_eia*). Rows for *id* are removed from *balancing_authority_eia861*.
- *reassign* (Optional[bool]): Whether to reassign utilities to parent BAs. Rows for *id* as BA in *balancing_authority_assn_eia861* are removed. Utilities assigned to *id* for a given year are reassigned to the BAs for which *id* is an associated utility.
- *replace* (Optional[bool]): Whether to remove rows where *id* is a utility in *balancing_authority_assn_eia861*. Applies only if *reassign=True*.

pudl.output.ferc714.**add_dates**(*rids_ferc714*, *report_dates*)

Broadcast respondent data across dates.

Parameters

- **rids_ferc714** (*pandas.DataFrame*) – A simple FERC 714 Respondent ID dataframe, without any date information.
- **report_dates** (*ordered collection of datetime*) – Dates for which each respondent should be given a record.

Raises **ValueError** – if a report_date column exists in rids_ferc714.

Returns Dataframe having all the same columns as the input rids_ferc714 with the addition of a report_date column, but with all records associated with each respondent_id_ferc714 duplicated on a per-date basis.

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*, *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.
- **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**(*pudl_engine*, *ds=None*, *freq=None*, *start_date=None*, *end_date=None*, *fill_fuel_cost=False*, *roll_fuel_cost=False*, *fill_net_gen=False*)

Bases: `object`

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

adjacency_ba_ferc714(*update=False*)
An interim FERC 714 output function.

advanced_metering_infrastructure_eia861(*update=False*)
An interim EIA 861 output function.

balancing_authority_assn_eia861(*update=False*)
An interim EIA 861 output function.

balancing_authority_eia861(*update=False*)
An interim EIA 861 output function.

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

demand_forecast_pa_ferc714(*update=False*)
An interim FERC 714 output function.

demand_hourly_pa_ferc714(*update=False*)
An interim FERC 714 output function.

demand_monthly_ba_ferc714(*update=False*)
An interim FERC 714 output function.

demand_response_eia861(*update=False*)
An interim EIA 861 output function.

demand_side_management_eia861(*update=False*)
An interim EIA 861 output function.

description_pa_ferc714(*update=False*)
An interim FERC 714 output function.

distributed_generation_eia861(*update=False*)
An interim EIA 861 output function.

distribution_systems_eia861(*update=False*)
An interim EIA 861 output function.

dynamic_pricing_eia861(*update=False*)
An interim EIA 861 output function.

energy_efficiency_eia861(*update=False*)
An interim EIA 861 output function.

etl_eia861(*update=False*)
A single function that runs the temporary EIA 861 ETL and sets all DFs.

This is an interim solution that provides a (somewhat) standard way of accessing the EIA 861 data prior to its being fully integrated into the PUDL database. If any of the dataframes is attempted to be accessed, all of them are set. Only the tables that have actual transform functions are included, and as new transform functions are completed, they would need to be added to the list below. Surely there is a way to do this automatically / magically but that's beyond my knowledge right now.

Parameters **update** (*bool*) – Whether to overwrite the existing dataframes if they exist.

etl_ferc714(*update=False*)
A single function that runs the temporary FERC 714 ETL and sets all DFs.

This is an interim solution, so that we can have a (relatively) standard way of accessing the FERC 714 data prior to getting it integrated into the PUDL DB. Some of these are not yet cleaned up, but there are dummy transform functions which pass through the raw DFs with some minor alterations, so all the data is available as it exists right now.

An attempt to access *any* of the dataframes results in all of them being populated, since generating all of them is almost the same amount of work as generating one of them.

Parameters **update** (*bool*) – Whether to overwrite the existing dataframes if they exist.

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_allocated_eia923(*update=False*)
Net generation from gen fuel table allocated to generators.

gen_eia923(*update=False*)
Pull EIA 923 net generation data by generator.

Net generation is reported in two seperate tables in EIA 923: in the generation_eia923 and generation_fuel_eia923 tables. While the generation_fuel_eia923 table is more complete (the generation_eia923 table includes only ~55% of the reported MWhs), the generation_eia923 table is more granular (it is reported at the generator level).

This method either grabs the generation_eia923 table that is reported by generator, or allocates net generation from the generation_fuel_eia923 table to the generator level.

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_original_eia923 (*update=False*)
Pull the original EIA 923 net generation data by generator.

gen_plants_ba_ferc714 (*update=False*)
An interim FERC 714 output function.

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

green_pricing_eia861 (*update=False*)
An interim EIA 861 output function.

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

id_certification_ferc714 (*update=False*)
An interim FERC 714 output function.

interchange_ba_ferc714 (*update=False*)
An interim FERC 714 output function.

lambda_description_ferc714 (*update=False*)
An interim FERC 714 output function.

lambda_hourly_ba_ferc714 (*update=False*)
An interim FERC 714 output function.

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 attibuted 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

mergers_eia861(*update=False*)
An interim EIA 861 output function.

net_energy_load_ba_ferc714(*update=False*)
An interim FERC 714 output function.

net_metering_eia861(*update=False*)
An interim EIA 861 output function.

non_net_metering_eia861(*update=False*)
An interim EIA 861 output function.

operational_data_eia861(*update=False*)
An interim EIA 861 output function.

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

plant_in_service_ferc1(*update=False*)
Pull the FERC Form 1 Plant in Service Table.

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_hydro_ferc1(*update=False*)
Pull the FERC Form 1 Hydro Plants Table.

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_pumped_storage_ferc1(*update=False*)
Pull the FERC Form 1 Pumped Storage Table.

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_small_ferc1(*update=False*)
Pull the FERC Form 1 Small Plants Table.

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

purchased_power_ferc1(*update=False*)
Pull the FERC Form 1 Purchased Power Table.

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

reliability_eia861(*update=False*)
An interim EIA 861 output function.

respondent_id_ferc714(*update=False*)
An interim FERC 714 output function.

sales_eia861(*update=False*)
An interim EIA 861 output function.

service_territory_eia861(*update=False*)
An interim EIA 861 output function.

utility_assn_eia861(*update=False*)
An interim EIA 861 output function.

utility_data_eia861(*update=False*)
An interim EIA 861 output function.

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 sqlitie 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

Code for transforming EIA data that pertains to more than one EIA Form.

This module helps normalize EIA datasets and infers additonal connections between EIA entities (i.e. utilities, plants, units, generators…). This includes:

- compiling a master list of plant, utility, boiler, and generator IDs that appear in any of the EIA 860 or 923 tables.
- inferring more complete boiler-generator associations.
- differentiating between static and time varying attributes associated with the EIA entities, storing the static fields with the entity table, and the variable fields in an annual table.

The boiler generator association inferrence (bga) takes the associations provided by the EIA 860, and expands on it using several methods which can be found in `pudl.transform.eia._boiler_generator_assn()`.

`pudl.transform.eia.`**`harvesting`**(*entity*, *eia_transformed_dfs*, *entities_dfs*, *eia860_ytd=False*, *debug=False*)

Compiles consistent records for various entities.

For each entity(plants, generators, boilers, utilties), this function finds all the harvestable columns from any table that they show up in. It then determines how consistent the records are and keeps the values that are mostly consistent. It compiles those consistent records into one normalized table.

There are a few things to note here. First being that we are not expecting the outcome here to be perfect! We choose to pull the most consistent record as reported across all the EIA tables and years, but we also required a "strictness" level of 70% (this is currently a hard coded argument for _occurrence_consistency). That means at least 70% of the records must be the same for us to use that value. So if values for an entity haven't been reported 70% consistently, then it will show up as a null value. We built in the ability to add special cases for columns where we want to apply a different method to, but the only ones we added was for latitude and longitude because they are by far the dirtiest.

We have determined which columns should be considered "static" or "annual". These can be found in constants in the *entities* dictionary. Static means That is should not change over time. Annual means there is annual variablity. This distinction was made in part by testing the consistency and in part by an understanding of how the entities and columns relate in the real world.

Parameters

- **entity** (*str*) – plants, generators, boilers, utilties
- **eia_transformed_dfs** (*dict*) – A dictionary of tbl names (keys) and transformed dfs (values)
- **entities_dfs** (*dict*) – A dictionary of entity table names (keys) and entity dfs (values)
- **eia860_ytd** (*boolean*) – if True, the etl run is attempting to include year-to-date updated from EIA 860M.
- **debug** (*bool*) – If True, this function will also return an additional dictionary of dataframes that includes the pre-deduplicated compiled records with the number of occurances of the entity and the record to see consistency of reported values.

Returns

A tuple containing: eia_transformed_dfs (dict): dictionary of tbl names (keys) and transformed dfs (values) entity_dfs (dict): dictionary of entity table names (keys) and entity dfs (values)

Return type tuple

Raises **AssertionError** – If the consistency of any record value is <90%.

Todo:

- Return to role of debug.
- Determine what to do with null records
- Determine how to treat mostly static records

`pudl.transform.eia.`**`transform`**(*eia_transformed_dfs*, *eia860_years=(2004, 2005, 2006, 2007, 2008, 2009, 2010, 2011, 2012, 2013, 2014, 2015, 2016, 2017, 2018, 2019)*, *eia923_years=(2009, 2010, 2011, 2012, 2013, 2014, 2015, 2016, 2017, 2018, 2019)*, *eia860_ytd=False*, *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 (kays) and transformed dataframes (values).
- **eia860_years** (*list*) – a list of years for EIA 860, must be continuous, and only include working years.
- **eia923_years** (*list*) – a list of years for EIA 923, must be continuous, and include only working years.
- **eia860_ytd** (*boolean*) – if True, the etl run is attempting to include year-to-date updated from EIA 860M.
- **debug** (*bool*) – if true, informational columns will be added into boiler_generator_assn

Returns two dictionaries having table names as keys and dataframes as values for the entity tables transformed EIA dataframes

Return type tuple

pudl.transform.eia860 module

Module to perform data cleaning functions on EIA860 data tables.

`pudl.transform.eia860.`**`OWNERSHIP_PLANT_GEN_ID_DUPES = [(56032, '1')]`**

EIA Plant IDs which have duplicate generators within the ownership table due to the removal of leading zeroes from the generator IDs.

Type tuple

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

Pull and transform the boilder generator association table.

Transformations include:

- Drop non-data rows with EIA notes.
- Drop duplicate rows.

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*`)`

Pull and transform the generators table.

There are three tabs that the generator records come from (proposed, existing, retired). Pre 2009, the existing and retired data are lumped together under a single generator file with one tab. We pull each tab into one dataframe and include an `operational_status` to indicate which tab the record came from. We use `operational_status` to parse the pre 2009 files as well.

Transformations include:

- Replace . values with NA.
- Update `operational_status_code` to reflect plant status as either proposed, existing or retired.
- Drop values with NA for plant and generator id.
- Replace 0 values with NA where appropriate.
- Convert Y/N/X values to boolean True/False.
- Convert U/Unknown values to NA.
- Map full spelling onto code values.
- Create a fuel_type_code_pudl field that organizes fuel types into clean, distinguishable categories.

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*`)`

Pull and transform the ownership table.

Transformations include:

- Replace . values with NA.

- Convert pre-2012 ownership percentages to proportions to match post-2012 reporting.

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.`**`plants`**(*eia860_dfs*, *eia860_transformed_dfs*)

Pull and transform 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.

Transformations include:

- Replace . values with NA.
- Homogenize spelling of county names.
- Convert Y/N/X values to boolean True/False.

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')*)

Transform 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*`)`

Pull and transform the utilities table.

Transformations include:

- Replace . values with NA.
- Fix typos in state abbreviations, convert to uppercase.
- Drop address_3 field (all NA).
- Combine phone number columns into one field and set values that don't mimic real US phone numbers to NA.
- Convert Y/N/X values to boolean True/False.
- Map full spelling onto code values.

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.eia861 module

Module to perform data cleaning functions on EIA861 data tables.

All transformations include: - Replace . values with NA.

`pudl.transform.eia861.`**`advanced_metering_infrastructure`**`(`*tfr_dfs*`)`

Transform the EIA 861 Advanced Metering Infrastructure table.

Transformations include:

- Tidy data by customer class.
- Drop total_meters columns (it's calculable with other fields).

Parameters **`tfr_dfs`** (*`dict`*) – A dictionary of transformed EIA 861 DataFrames, keyed by table name. It will be mutated by this function.

Returns A dictionary of transformed EIA 861 dataframes, keyed by table name.

Return type dict

`pudl.transform.eia861.`**`balancing_authority`**`(`*tfr_dfs*`)`

Transform the EIA 861 Balancing Authority table.

Transformations include:

- Fill in balancing authrority IDs based on date, utility ID, and BA Name.
- Backfill balancing authority codes based on BA ID.
- Fix BA code and ID typos.

Parameters **tfr_dfs** (*dict*) – A dictionary of transformed EIA 861 DataFrames, keyed by table name. It will be mutated by this function.

Returns A dictionary of transformed EIA 861 dataframes, keyed by table name.

Return type dict

`pudl.transform.eia861.`**`balancing_authority_assn`**`(`*tfr_dfs*`)`

Compile a balancing authority, utility, state association table.

For the years up through 2012, the only BA-Util information that's available comes from the balancing_authority_eia861 table, and it does not include any state-level information. However, there is utility-state association information in the sales_eia861 and other data tables.

For the years from 2013 onward, there's explicit BA-Util-State information in the data tables (e.g. sales_eia861). These observed associations can be compiled to give us a picture of which BA-Util-State associations exist. However, we need to merge in the balancing authority IDs since the data tables only contain the balancing authority codes.

Parameters **tfr_dfs** (*dict*) – A dictionary of transformed EIA 861 dataframes. This must include any dataframes from which we want to compile BA-Util-State associations, which means this function has to be called after all the basic transformfunctions that depend on only a single raw table.

Returns a dictionary of transformed dataframes. This function both compiles the association table, and finishes the normalization of the balancing authority table. It may be that once the harvesting process incorporates the EIA 861, some or all of this functionality should be pulled into the phase-2 transform functions.

Return type dict

`pudl.transform.eia861.`**`demand_response`**`(`*tfr_dfs*`)`

Transform the EIA 861 Demand Response table.

Transformations include:

- Fill in NA balancing authority codes with UNK (because it's part of the primary key).
- Tidy subset of the data by customer class.
- Drop duplicate rows based on primary keys.
- Convert 1000s of dollars into dollars.

Parameters **tfr_dfs** (*dict*) – A dictionary of transformed EIA 861 DataFrames, keyed by table name. It will be mutated by this function.

Returns A dictionary of transformed EIA 861 dataframes, keyed by table name.

Return type dict

`pudl.transform.eia861.`**`demand_side_management`**`(`*tfr_dfs*`)`

Transform the EIA 861 Demand Side Management table.

In 2013, the EIA changed the contents of the 861 form so that information pertaining to demand side management was no longer housed in a single table, but rather two seperate ones pertaining to energy efficiency and demand response. While the pre and post 2013 tables contain similar information, one column in the pre-2013 demand side management table may not have an obvious column equivalent in the post-2013 energy efficiency or demand response data. We've addressed this by keeping the demand side management and energy efficiency and demand response tables seperate. Use the DSM table for pre 2013 data and the EE / DR tables for post 2013 data. Despite the uncertainty of comparing across these years, the data are similar and we hope to provide a cohesive dataset in the future with all years and comprable columns combined.

Transformations include:

- Clean up NERC codes and ensure one per row.
- Remove demand_side_management and data_observed columns (they are all the same).
- Tidy subset of the data by customer class.
- Convert Y/N columns to booleans.
- Convert 1000s of dollars into dollars.

Parameters **tfr_dfs** (*dict*) – A dictionary of transformed EIA 861 DataFrames, keyed by table name. It will be mutated by this function.

Returns A dictionary of transformed EIA 861 dataframes, keyed by table name.

Return type dict

`pudl.transform.eia861.`**`distributed_generation`**(*tfr_dfs*)

Transform the EIA 861 Distributed Generation table.

Transformations include:

- Map full spelling onto code values.
- Convert pre-2010 percent values in mw values.
- Remove total columns calculable with other fields.
- Tidy subset of the data by tech class.
- Tidy subset of the data by fuel class.

Parameters **tfr_dfs** (*dict*) – A dictionary of transformed EIA 861 DataFrames, keyed by table name. It will be mutated by this function.

Returns A dictionary of transformed EIA 861 dataframes, keyed by table name.

Return type dict

`pudl.transform.eia861.`**`distribution_systems`**(*tfr_dfs*)

Transform the EIA 861 Distribution Systems table.

Transformations include:

- No additional transformations.

Parameters **tfr_dfs** (*dict*) – A dictionary of transformed EIA 861 DataFrames, keyed by table name. It will be mutated by this function.

Returns A dictionary of transformed EIA 861 dataframes, keyed by table name.

Return type dict

`pudl.transform.eia861.`**`dynamic_pricing`**(*tfr_dfs*)

Transform the EIA 861 Dynamic Pricing table.

Transformations include:

- Tidy subset of the data by customer class.
- Convert Y/N columns to booleans.

Parameters **tfr_dfs** (*dict*) – A dictionary of transformed EIA 861 DataFrames, keyed by table name. It will be mutated by this function.

Returns A dictionary of transformed EIA 861 dataframes, keyed by table name.

Return type dict

`pudl.transform.eia861.`**`energy_efficiency`**(*tfr_dfs*)

Transform the EIA 861 Energy Efficiency table.

Transformations include:

- Tidy subset of the data by customer class.
- Drop website column (almost no valid information).
- Convert 1000s of dollars into dollars.

Parameters **tfr_dfs** (*dict*) – A dictionary of transformed EIA 861 DataFrames, keyed by table name. It will be mutated by this function.

Returns A dictionary of transformed EIA 861 dataframes, keyed by table name.

Return type dict

`pudl.transform.eia861.`**`green_pricing`**(*tfr_dfs*)

Transform the EIA 861 Green Pricing table.

Transformations include:

- Tidy subset of the data by customer class.
- Convert 1000s of dollars into dollars.

Parameters **tfr_dfs** (*dict*) – A dictionary of transformed EIA 861 DataFrames, keyed by table name. It will be mutated by this function.

Returns A dictionary of transformed EIA 861 dataframes, keyed by table name.

Return type dict

`pudl.transform.eia861.`**`mergers`**(*tfr_dfs*)

Transform the EIA 861 Mergers table.

Transformations include:

- Map full spelling onto code values.
- Retain preceeding zeros in zipcode field.

Parameters **tfr_dfs** (*dict*) – A dictionary of transformed EIA 861 DataFrames, keyed by table name. It will be mutated by this function.

Returns A dictionary of transformed EIA 861 dataframes, keyed by table name.

Return type dict

`pudl.transform.eia861.`**`net_metering`**(*tfr_dfs*)

Transform the EIA 861 Net Metering table.

Transformations include:

- Remove rows with utility ids 99999.

- Tidy subset of the data by customer class.
- Tidy subset of the data by tech class.

Parameters **tfr_dfs** (*dict*) – A dictionary of transformed EIA 861 DataFrames, keyed by table name. It will be mutated by this function.

Returns A dictionary of transformed EIA 861 dataframes, keyed by table name.

Return type dict

`pudl.transform.eia861.`**`non_net_metering`**(*tfr_dfs*)

Transform the EIA 861 Non-Net Metering table.

Transformations include:

- Remove rows with utility ids 99999.
- Drop duplicate rows.
- Tidy subset of the data by customer class.
- Tidy subset of the data by tech class.

Parameters **tfr_dfs** (*dict*) – A dictionary of transformed EIA 861 DataFrames, keyed by table name. It will be mutated by this function.

Returns A dictionary of transformed EIA 861 dataframes, keyed by table name.

Return type dict

`pudl.transform.eia861.`**`normalize_balancing_authority`**(*tfr_dfs*)

Finish the normalization of the balancing_authority_eia861 table.

The balancing_authority_assn_eia861 table depends on information that is only available in the UN-normalized form of the balancing_authority_eia861 table, so and also on having access to a bunch of transformed data tables, so it can compile the observed combinations of report dates, balancing authorities, states, and utilities. This means that we have to hold off on the final normalization of the balancing_authority_eia861 table until the rest of the transform process is over.

`pudl.transform.eia861.`**`operational_data`**(*tfr_dfs*)

Transform the EIA 861 Operational Data table.

Transformations include:

- Remove rows with utility ids 88888.
- Remove rows with NA utility id.
- Clean up NERC codes and ensure one per row.
- Convert data_observed field I/O into boolean.
- Tidy subset of the data by revenue class.
- Convert 1000s of dollars into dollars.

Parameters **tfr_dfs** (*dict*) – A dictionary of transformed EIA 861 DataFrames, keyed by table name. It will be mutated by this function.

Returns A dictionary of transformed EIA 861 dataframes, keyed by table name.

Return type dict

`pudl.transform.eia861.reliability(tfr_dfs)`

Transform the EIA 861 Reliability table.

Transformations include:

- Tidy subset of the data by reliability standard.
- Convert Y/N columns to booleans.
- Map full spelling onto code values.
- Drop duplicate rows.

Parameters **tfr_dfs** (*dict*) – A dictionary of transformed EIA 861 DataFrames, keyed by table name. It will be mutated by this function.

Returns A dictionary of transformed EIA 861 dataframes, keyed by table name.

Return type dict

`pudl.transform.eia861.sales(tfr_dfs)`

Transform the EIA 861 Sales table.

Transformations include:

- Remove rows with utility ids 88888 and 99999.
- Tidy data by customer class.
- Drop primary key duplicates.
- Convert 1000s of dollars into dollars.
- Convert data_observed field I/O into boolean.
- Map full spelling onto code values.

`pudl.transform.eia861.service_territory(tfr_dfs)`

Transform the EIA 861 utility service territory table.

Transformations include:

- Homogenize spelling of county names.
- Add field for state/county FIPS code.

Parameters **tfr_dfs** (*dict*) – A dictionary of DataFrame objects in which pages from EIA861 form (keys) correspond to normalized DataFrames of values from that page (values).

Returns

a dictionary of pandas.DataFrame objects in which pages from EIA861 form (keys) correspond to normalized DataFrames of values from that page (values).

Return type dict

`pudl.transform.eia861.`**`transform`**(*raw_dfs*, *eia861_tables=('service_territory_eia861', 'balancing_authority_eia861', 'sales_eia861', 'advanced_metering_infrastructure_eia861', 'demand_response_eia861', 'demand_side_management_eia861', 'distributed_generation_eia861', 'distribution_systems_eia861', 'dynamic_pricing_eia861', 'energy_efficiency_eia861', 'green_pricing_eia861', 'mergers_eia861', 'net_metering_eia861', 'non_net_metering_eia861', 'operational_data_eia861', 'reliability_eia861', 'utility_data_eia861')*)

Transform EIA 861 DataFrames.

Parameters

- **`raw_dfs`** (*dict*) – a dictionary of tab names (keys) and DataFrames (values). This can be generated by pudl.
- **`eia861_tables`** (*tuple*) – A tuple containing the names of the EIA 861 tables that can be pulled into PUDL.

Returns A dictionary of DataFrame objects in which pages from EIA 861 form (keys) corresponds to a normalized DataFrame of values from that page (values).

Return type dict

`pudl.transform.eia861.`**`utility_assn`**(*tfr_dfs*)

Harvest a Utility-Date-State Association Table.

`pudl.transform.eia861.`**`utility_data`**(*tfr_dfs*)

Transform the EIA 861 Utility Data table.

Transformations include:

- Remove rows with utility ids 88888.
- Clean up NERC codes and ensure one per row.
- Tidy subset of the data by NERC region.
- Tidy subset of the data by RTO.
- Convert Y/N columns to booleans.

Parameters **`tfr_dfs`** (*dict*) – A dictionary of transformed EIA 861 DataFrames, keyed by table name. It will be mutated by this function.

Returns A dictionary of transformed EIA 861 dataframes, keyed by table name.

Return type dict

pudl.transform.eia923 module

Module to perform data cleaning functions on EIA923 data tables.

`pudl.transform.eia923.`**`boiler_fuel`**(*eia923_dfs*, *eia923_transformed_dfs*)

Transforms the boiler_fuel_eia923 table.

Transformations include:

- Remove fields implicated elsewhere.
- Drop values with plant and boiler id values of NA.

- Replace . values with NA.
- Create a fuel_type_code_pudl field that organizes fuel types into clean, distinguishable categories.
- Combine year and month columns into a single date column.

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.

Transformations include:

- Remove fields implicated elsewhere.
- Drop duplicates with MSHA ID.

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.

Transformations include:

- Remove fields implicated elsewhere.
- Replace . values with NA.
- Standardize codes values.
- Fix dates.
- Replace invalid mercury content values with NA.

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.

Transformations include:

- Drop rows with NA for generator id.
- Remove fields implicated elsewhere.
- Replace . values with NA.
- Drop generator-date row duplicates (all have no data).

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.

Transformations include:

- Remove fields implicated elsewhere.
- Replace . values with NA.
- Remove rows with utility ids 99999.
- Create a fuel_type_code_pudl field that organizes fuel types into clean, distinguishable categories.
- Combine year and month columns into a single date column.

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.

Transformations include:

- Map full spelling onto code values.
- Convert Y/N columns to booleans.
- Remove excess white space around values.
- Drop duplicate rows.

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 with table names as keys and `pandas.DataFrame` objects as values, where the contents of the DataFrames correspond to cleaned and normalized PUDL database tables, ready for loading.

Return type dict

pudl.transform.epacems module

Module to perform data cleaning functions on EPA CEMS data tables.

`pudl.transform.epacems.`**`add_facility_id_unit_id_epa`**(*df*)

Harmonize columns that are added later.

The datapackage validation checks for consistent column names, and these two columns aren't present before August 2008, so this adds them in.

Parameters **df** (*pandas.DataFrame*) – A CEMS dataframe

Returns The same DataFrame guaranteed to have int facility_id and unit_id_epa cols.

Return type pandas.Dataframe

`pudl.transform.epacems.`**`correct_gross_load_mw`**(*df*)

Fix values of gross load that are wrong by orders of magnitude.

Parameters **df** (*pandas.DataFrame*) – A CEMS dataframe

Returns The same DataFrame with corrected gross load values.

Return type pandas.DataFrame

`pudl.transform.epacems.`**`fix_up_dates`**(*df*, *plant_utc_offset*)

Fix the dates for the CEMS data.

Transformations include:

- Account for timezone differences with offset from UTC.

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. EPA has compiled a crosswalk that maps one set of IDs to the other, but we haven't integrated it yet. It can be found at:

https://github.com/USEPA/camd-eia-crosswalk

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: Actually implement the function…

`pudl.transform.epacems.`**`transform`**(*epacems_raw_dfs*, *datapkg_dir*)

Transform EPA CEMS hourly data for use in datapackage export.

Todo: 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.

pudl.transform.ferc1.**CONSTRUCTION_TYPE_STRINGS = {'conventional': ['conventional', 'conver**
A dictionary of construction types (keys) and lists of construction type strings associated with each type (values) from FERC Form 1.

There are many strings that weren't categorized, including crosses between conventional and outdoor, PV, wind, combined cycle, and internal combustion. The lists are broken out into the two types specified in Form 1: conventional and outdoor. These lists are inclusive so that variants of conventional (e.g. "conventional full") and outdoor (e.g. "outdoor full" and "outdoor hrsg") are included.

Type dict

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.**FUEL_STRINGS = {'coal': ['coal', 'coal-subbit', 'lignite', 'coal(sb)**
A mapping a canonical fuel name to a list of strings which are used to represent that fuel in the FERC Form 1 Reporting. Case is ignored, as all fuel strings are converted to a lower case in the data set.

Type dict

pudl.transform.ferc1.**FUEL_UNIT_STRINGS = {'bbl': ['barrel', 'bbls', 'bbl', 'barrels', 'bb**
A dictionary linking fuel units (keys) to lists of various strings representing those fuel units (values)

Type dict

pudl.transform.ferc1.**PLANT_KIND_STRINGS = {'combined_cycle': ['Combined cycle', 'combined**
A mapping from canonical plant kinds (keys) to the associated freeform strings (values) identified as being associated with that kind of plant in the FERC Form 1 raw data. There are many strings that weren't categorized, Solar and Solar Project were not classified as these do not indicate if they are solar thermal or photovoltaic. Variants on Steam (e.g. "steam 72" and "steam and gas") were classified based on additional research of the plants on the Internet.

Type dict

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.**cols_to_cats**(*df*, *cat_name*, *col_cats*)
Turn top-level MultiIndex columns into a categorial column.

In some cases FERC Form 1 data comes with many different types of related values interleaved in the same table – e.g. current year and previous year income – this can result in DataFrames that are hundreds of columns wide, which is unwieldy. This function takes those top level MultiIndex labels and turns them into categories in a single column, which can be used to select a particular type of report.

Parameters

- **df** (*pandas.DataFrame*) – the dataframe to be simplified.
- **cat_name** (*str*) – the label of the column to be created indicating what MultiIndex label the values came from.
- **col_cats** (*dict*) – a dictionary with top level MultiIndex labels as keys, and the category to which they should be mapped as values.

Returns A re-shaped/re-labeled dataframe with one fewer levels of MultiIndex in the columns, and an additional column containing the assigned labels.

Return type pandas.DataFrame

`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 & lower). Fuel and fuel unit strings are also standardized using our cleanstrings() function and string cleaning dictionaries found above (FUEL_STRINGS, etc.)

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 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.

Return type pandas.DataFrame

Raises **AssertionError** – If the DataFrame input does not have the columns required to run the function.

`pudl.transform.ferc1.make_ferc1_clf`(*plants_df*, *ngram_min=2*, *ngram_max=10*, *min_sim=0.75*, *plant_name_ferc1_wt=2.0*, *plant_type_wt=2.0*, *construction_type_wt=1.0*, *capacity_mw_wt=1.0*, *construction_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_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.
- **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.

Re-organizes the original FERC Form 1 Plant in Service data by unpacking the rows as needed on a year by year basis, to organize them into columns. The "columns" in the original FERC Form 1 denote starting balancing, ending balance, additions, retirements, adjustments, and transfers – these categories are turned into labels in a column called "amount_type". Because each row in the transformed table is composed of many individual records (rows) from the original table, row_number can't be part of the record_id, which means they are no longer unique. To infer exactly what record a given piece of data came from, the record_id and the row_map (found in the PUDL package_data directory) can be used.

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 electricty 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_steam_ferc1', 'plants_small_ferc1', 'plants_hydro_ferc1', 'plants_pumped_storage_ferc1', 'purchased_power_ferc1', 'plant_in_service_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

pudl.transform.ferc1.**unpack_table**(*ferc1_df*, *table_name*, *data_cols*, *data_rows*)

Normalize a row-and-column based FERC Form 1 table.

Pulls the named database table from the FERC Form 1 DB and uses the corresponding ferc1_row_map to unpack the row_number coded data.

Parameters

- **ferc1_df** (*pandas.DataFrame*) – Raw FERC Form 1 DataFrame from the DB.
- **table_name** (*str*) – Original name of the FERC Form 1 DB table.
- **data_cols** (*list*) – List of strings corresponding to the original FERC Form 1 database table column labels – these are the columns of data that we are extracting (it can be a subset of the columns which are present in the original database).
- **data_rows** (*list*) – List of row_names to extract, as defined in the FERC 1 row maps. Set to slice(None) if you want all rows.

Returns pandas.DataFrame

pudl.transform.ferc714 module

Transformation of the FERC Form 714 data.

pudl.transform.ferc714.**BAD_RESPONDENTS = [319, 99991, 99992, 99993, 99994, 99995]**

Fake respondent IDs for database test entities.

pudl.transform.ferc714.**EIA_CODE_FIXES = {125: 2775, 134: 5416, 203: 12341, 257: 59504,**

Overrides of FERC 714 respondent IDs with wrong or missing EIA Codes

pudl.transform.ferc714.**OFFSET_CODES = {'AKDT': Timedelta('-1 days +15:00:00'), 'AKST': Time**

A mapping of timezone offset codes to Timedelta offsets from UTC.

from one year to the next, and these result in duplicate records, which are Note that the FERC 714 instructions state that all hourly demand is to be reported in STANDARD time for whatever timezone is being used. Even though many respondents use daylight savings / standard time abbreviations, a large majority do appear to conform to using a single UTC offset throughout the year. There are 6 instances in which the timezone associated with reporting changed dropped.

pudl.transform.ferc714.**TZ_CODES = {'AKDT': 'America/Anchorage', 'AKST': 'America/Anchorage**
Mapping between standardized time offset codes and canonical timezones.

pudl.transform.ferc714.**adjacency_ba**(*tfr_dfs*)
A stub transform function.

pudl.transform.ferc714.**demand_forecast_pa**(*tfr_dfs*)
A stub transform function.

pudl.transform.ferc714.**demand_hourly_pa**(*tfr_dfs*)
Transform the hourly demand time series by Planning Area.

Transformations include:

- Clean UTC offset codes.
- Replace UTC offset codes with UTC offset and timezone.
- Drop 25th hour rows.
- Set records with 0 UTC code to 0 demand.
- Drop duplicate rows.
- Flip negative signs for reported demand.

Parameters **tfr_dfs** (*dict*) – A dictionary of (partially) transformed dataframes, to be cleaned up.

Returns The input dictionary of dataframes, but with a finished pa_demand_hourly_ferc714 dataframe.

Return type dict

pudl.transform.ferc714.**demand_monthly_ba**(*tfr_dfs*)
A stub transform function.

pudl.transform.ferc714.**description_pa**(*tfr_dfs*)
A stub transform function.

pudl.transform.ferc714.**gen_plants_ba**(*tfr_dfs*)
A stub transform function.

pudl.transform.ferc714.**id_certification**(*tfr_dfs*)
A stub transform function.

pudl.transform.ferc714.**interchange_ba**(*tfr_dfs*)
A stub transform function.

pudl.transform.ferc714.**lambda_description**(*tfr_dfs*)
A stub transform function.

pudl.transform.ferc714.**lambda_hourly_ba**(*tfr_dfs*)
A stub transform function.

pudl.transform.ferc714.**net_energy_load_ba**(*tfr_dfs*)
A stub transform function.

pudl.transform.ferc714.**respondent_id**(*tfr_dfs*)
Transform the FERC 714 respondent IDs, names, and EIA utility IDs.

This consists primarily of dropping test respondents and manually assigning EIA utility IDs to a few FERC Form 714 respondents that report planning area demand, but which don't have their corresponding EIA utility IDs provided by FERC for some reason (including PacifiCorp).

Parameters **tfr_dfs** (*dict*) – A dictionary of (partially) transformed dataframes, to be cleaned up.

Returns The input dictionary of dataframes, but with a finished respondent_id_ferc714 dataframe.

Return type dict

pudl.transform.ferc714.**transform**(*raw_dfs*, *tables=('respondent_id_ferc714', 'id_certification_ferc714', 'gen_plants_ba_ferc714', 'demand_monthly_ba_ferc714', 'net_energy_load_ba_ferc714', 'adjacency_ba_ferc714', 'interchange_ba_ferc714', 'lambda_hourly_ba_ferc714', 'lambda_description_ferc714', 'description_pa_ferc714', 'demand_forecast_pa_ferc714', 'demand_hourly_pa_ferc714')*)

Transform the raw FERC 714 dataframes into datapackage ready ouputs.

Parameters

- **raw_dfs** (*dict*) – A dictionary of raw pandas.DataFrame objects, as read out of the original FERC 714 CSV files. Generated by the *pudl.extract.ferc714.extract()* function.
- **tables** (*iterable*) – The set of PUDL tables within FERC 714 that we should process. Typically set to all of them, unless

Returns A dictionary of pandas.DataFrame objects that are ready to be output in a data package / database table.

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 :ref: *data-sources*. This process begins with a dictionary of "raw" `pandas.DataFrame` objects produced by the corresponding data source specific routines from the *pudl.extract* subpackage, and ends with a dictionary of `pandas.DataFrame` objects that are fully normalized, cleaned, and congruent with the tabular datapackage metadata – i.e. they are ready to be exported by the *pudl.load* module.

Inputs to the transform functions are a dictionary of dataframes, each of which represents a concatenation of records with common column names from across some set of years of reported data. The names of those columns are determined by the xlsx_maps metadata associated with the given dataset in PUDL's package_metadata.

This raw data is transformed in 3 main steps:

1. Structural transformations that re-shape / tidy the data and turn it into rows that represent a single observation, and columns that represent a single variable. These transformations should not require knowledge of or access to the contents of the data, which may or may not yet be usable at this point, depending on the true data type and how much cleaning has to happen. One exception to this that may come up is the need to clean up columns that are part of the primary composite key, since you can't usefully index on NA values. Alternatively this might mean removing rows that have invalid key values.
2. Data type compatibility: whatever massaging of the data is required to ensure that it can be cast to the appropriate data type, including identifying NA values and assigning them to an appropriate type-specific NA value. At the end of this you can assign all the columns their (preferably nullable) types. Note that because some of the columns that exist at this point may not end up in the final database table, you may need to set them individually, rather than using the systemwide dictionary of column data types.
3. Value based data cleaning: At this point every column should have a known, homogenous type, allowing it to be reliably manipulated as a Series, so we can move on to cleaning up the values themselves. This includes

re-coding freeform string fields to impose a controlled vocabulary, converting column units (e.g. kWh to MWh) and renaming the columns appropriately, as well as correcting clear data entry errors.

At the end of the main coordinating transform() function, every column that remains in each of the transformed dataframes should correspond to a column that will exist in the database and be associated with the EIA datasets, which means it is also part of the EIA column namespace. It's important that you make sure these column names match the naming conventions that are being used, and if any of the columns exist in other tables, that they have exactly the same name and datatype.

If you find that you need to rename a column for it to conform to those requirements, in many cases that should happen in the xlsx_map metadata, so that column renamings can be kept to a minimum and only used for real semantic transformations of a column (like a unit conversion).

At the end of this step, it should be easy to categorize every column in every dataframe as to whether it is a "data" column (containing data unique to the table it is found in) or whether it is part of the primary key for the table (the minimal set of columns whose values are required to uniquely specify a record), and/or whether it is a "denormalized" column whose home table is really elsewhere in the database. Note that denormalized columns may also be part of the primary key. This information is important for the step after the intra-table transformations during which the collection of EIA tables is normalized as a whole.

pudl.workspace package

Submodules

pudl.workspace.datastore module

Datastore manages file retrieval for PUDL datasets.

exception `pudl.workspace.datastore.`**`CheckumMismatch`**

Bases: `ValueError`

Resource checksum (md5) does not match.

class `pudl.workspace.datastore.`**`DatapackageDescriptor`**(*datapackage_json: dict, dataset: str, doi: str*)

Bases: `object`

A simple wrapper providing access to datapackage.json contents.

`get_json_string`() → str

Exports the underlying json as normalized (sorted, indented) json string.

`get_partitions`(*name: Optional[str] = None*) → Dict[str, Set[str]]

Returns mapping of all known partition keys to the set of its known values.

`get_resource_path`(*name: str*) → str

Returns zenodo url that holds contents of given named resource.

`get_resources`(*name: Optional[str] = None, **filters: Any*) → Iterator[*pudl.workspace.resource_cache.PudlResourceKey*]

Returns series of PudlResourceKey identifiers for matching resources.

Parameters

- **name** (*str*) – if specified, find resource(s) with this name.
- **filters** (*dict*) – if specified, find resoure(s) matching these key=value constraints. The constraints are matched against the 'parts' field of the resource entry in the datapackage.json.

validate_checksum(*name: str, content: str*) → bool
Returns True if content matches checksum for given named resource.

class pudl.workspace.datastore.**Datastore**(*local_cache_path: Optional[pathlib.Path] = None, gcs_cache_path: Optional[str] = None, sandbox: bool = False, timeout: float = 15*)
Bases: `object`

Handle connections and downloading of Zenodo Source archives.

get_datapackage_descriptor(*dataset: str*) → pudl.workspace.datastore.DatapackageDescriptor
Fetch datapackage descriptor for given dataset either from cache or from zenodo.

get_known_datasets() → List[str]
Returns list of supported datasets.

get_resources(*dataset: str, cached_only: bool = False, skip_optimally_cached: bool = False, **filters: Any*) → Iterator[Tuple[pudl.workspace.resource_cache.PudlResourceKey, bytes]]
Return content of the matching resources.

Parameters

- **dataset** (*str*) – name of the dataset to query.
- **cached_only** (*bool*) – if True, only retrieve resources that are present in the cache.
- **skip_optimally_cached** (*bool*) – if True, only retrieve resources that are not optimally cached. This triggers attempt to optimally cache these resources.
- **filters** (*key=val*) – only return resources that match the key-value mapping in their
- **metadata["parts"]** –

Yields (PudlResourceKey, io.BytesIO) holding content for each matching resource

get_unique_resource(*dataset: str, **filters: Any*) → bytes
Returns content of a resource assuming there is exactly one that matches.

get_zipfile_resource(*dataset: str, **filters: Any*) → zipfile.ZipFile
Retrieves unique resource and opens it as a ZipFile.

remove_from_cache(*res: pudl.workspace.resource_cache.PudlResourceKey*)
Remove given resource from the associated cache.

class pudl.workspace.datastore.**ParseKeyValues**(*option_strings, dest, nargs=None, const=None, default=None, type=None, choices=None, required=False, help=None, metavar=None*)
Bases: `argparse.Action`

Transforms k1=v1,k2=v2,… into dict(k1=v1, k2=v2, …).

class pudl.workspace.datastore.**ZenodoFetcher**(*sandbox: bool = False, timeout: float = 15.0*)
Bases: `object`

API for fetching datapackage descriptors and resource contents from zenodo.

API_ROOT = {'production': 'https://zenodo.org/api', 'sandbox': 'https://sandbox.zeno

DOI = {'production': {'censusdp1tract': '10.5281/zenodo.4127049', 'eia860': '10.528

TOKEN = {'production': 'KXcG5s9TqeuPh1Ukt5QYbzhCElp9LxuqAuiwdqHP0WS4qGIQiydHn6FBtdJ5'

get_descriptor(*dataset: str*) → *pudl.workspace.datastore.DatapackageDescriptor*
Returns DatapackageDescriptor for given dataset.

get_doi(*dataset: str*) → str
Returns DOI for given dataset.

get_known_datasets() → List[str]
Returns list of supported datasets.

get_resource(*res: pudl.workspace.resource_cache.PudlResourceKey*) → bytes
Given resource key, retrieve contents of the file from zenodo.

get_resource_key(*dataset: str, name: str*) → *pudl.workspace.resource_cache.PudlResourceKey*
Returns PudlResourceKey for given resource.

pudl.workspace.datastore.**fetch_resources**(*dstore: pudl.workspace.datastore.Datastore, datasets: List[str], args: argparse.Namespace*) → None
Retrieve all matching resources and store them in the cache.

pudl.workspace.datastore.**main**()
Cache datasets.

pudl.workspace.datastore.**parse_command_line**()
Collect the command line arguments.

pudl.workspace.datastore.**print_partitions**(*dstore: pudl.workspace.datastore.Datastore, datasets: List[str]*) → None
Prints known partition keys and its values for each of the datasets.

pudl.workspace.datastore.**validate_cache**(*dstore: pudl.workspace.datastore.Datastore, datasets: List[str], args: argparse.Namespace*) → None
Validate elements in the datastore cache. Delete invalid entires from cache.

pudl.workspace.resource_cache module

Implementations of datastore resource caches.

class pudl.workspace.resource_cache.**AbstractCache**(*read_only: bool = False*)
Bases: `abc.ABC`

Defines interaface for the generic resource caching layer.

abstract add(*resource: pudl.workspace.resource_cache.PudlResourceKey, content: bytes*) → None
Adds resource to the cache and sets the content.

abstract contains(*resource: pudl.workspace.resource_cache.PudlResourceKey*) → bool
Returns True if the resource is present in the cache.

abstract delete(*resource: pudl.workspace.resource_cache.PudlResourceKey*) → None
Removes the resource from cache.

abstract get(*resource: pudl.workspace.resource_cache.PudlResourceKey*) → bytes
Retrieves content of given resource or throws KeyError.

is_read_only() → bool
Returns true if the cache is read-only and should not be modified.

class pudl.workspace.resource_cache.**GoogleCloudStorageCache**(*gcs_path: str, **kwargs: Any*)
Bases: `pudl.workspace.resource_cache.AbstractCache`

Implements file cache backed by Google Cloud Storage bucket.

add(*resource:* pudl.workspace.resource_cache.PudlResourceKey, *value:* *bytes*)
Adds (or updates) resource to the cache with given value.

contains(*resource:* pudl.workspace.resource_cache.PudlResourceKey) → bool
Returns True if resource is present in the cache.

delete(*resource:* pudl.workspace.resource_cache.PudlResourceKey)
Deletes resource from the cache.

get(*resource:* pudl.workspace.resource_cache.PudlResourceKey) → bytes
Retrieves value associated with given resource.

class pudl.workspace.resource_cache.**LayeredCache**(**caches: List[*pudl.workspace.resource_cache.AbstractCache*]*, ***kwargs: Any*)
Bases: *pudl.workspace.resource_cache.AbstractCache*

Implements multi-layered system of caches.

This allows building multi-layered system of caches. The idea is that you can have faster local caches with fall-back to the more remote or expensive caches that can be acessed in case of missing content.

Only the closest layer is being written to (set, delete), while all remaining layers are read-only (get).

add(*resource:* pudl.workspace.resource_cache.PudlResourceKey, *value*)
Adds (or replaces) resource into the cache with given value.

add_cache_layer(*cache:* pudl.workspace.resource_cache.AbstractCache)
Adds caching layer. The priority is below all other.

contains(*resource:* pudl.workspace.resource_cache.PudlResourceKey) → bool
Returns True if resource is present in the cache.

delete(*resource:* pudl.workspace.resource_cache.PudlResourceKey)
Removes resource from the cache if the cache is not in the read_only mode.

get(*resource:* pudl.workspace.resource_cache.PudlResourceKey) → bytes
Returns content of a given resource.

is_optimally_cached(*resource:* pudl.workspace.resource_cache.PudlResourceKey) → bool
Returns true if the resource is contained in the closest write-enabled layer.

num_layers()
Returns number of caching layers that are in this LayeredCache.

class pudl.workspace.resource_cache.**LocalFileCache**(*cache_root_dir:* *pathlib.Path*, ***kwargs: Any*)
Bases: *pudl.workspace.resource_cache.AbstractCache*

Simple key-value store mapping PudlResourceKeys to ByteIO contents.

add(*resource:* pudl.workspace.resource_cache.PudlResourceKey, *content:* *bytes*)
Adds (or updates) resource to the cache with given value.

contains(*resource:* pudl.workspace.resource_cache.PudlResourceKey) → bool
Returns True if resource is present in the cache.

delete(*resource:* pudl.workspace.resource_cache.PudlResourceKey)
Deletes resource from the cache.

get(*resource:* pudl.workspace.resource_cache.PudlResourceKey) → bytes
Retrieves value associated with a given resource.

class `pudl.workspace.resource_cache.`**`PudlResourceKey`**(*dataset: str*, *doi: str*, *name: str*)
Bases: `tuple`

Uniquely identifies a specific resource.

`dataset: str`
Alias for field number 0

`doi: str`
Alias for field number 1

`get_local_path`() → pathlib.Path
Returns (relative) path that should be used when caching this resource.

`name: str`
Alias for field number 2

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 data management 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 –clobber option.

The directory structure set up for PUDL looks like this:

PUDL_IN

└── **data** ├── censusdp1tract ├── eia860 ├── eia860m ├── eia861 ├── eia923 ├── epacems ├── ferc1 ├── ferc714 └── tmp

PUDL_OUT ├── datapkg ├── parquet ├── settings └── sqlite

Initially, the directories in the data store will be empty. The pudl_datastore 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 form 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_datastore`. 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 datapacakges 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 datapacakges will be generated. If the settings include a datapackage that has empty parameters, the other valid datatpackages will be generated, but not the empty one. If there are invalid parameters (meaning a partition that is not included in the pudl.constant.working_partitions), the build will fail early on in the process.

The datapackages will be stored in "PUDL_OUT" in the "datapackge" 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 initilize 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.TRANSIT_TYPE_DICT = {'CV': 'conveyer', 'PL': 'pipeline', 'RR': 'railroad',`
A dictionary of datasets (keys) and keywords (values).

Type dict

`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.`**`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': 'unk`**
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.data_source_info = {'eia860': {'path': 'https://www.eia.gov/electricity/da`
A dictionary of dictionaries containing datasources (keys) and associated attributes (values)

Type dict

`pudl.constants.data_sources = ('eia860', 'eia861', 'eia923', 'epacems', 'epaipm', 'ferc1',`
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 enumerating EIA 860 tables for which PUDL's ETL works.

Type tuple

`pudl.constants.eia923_pudl_tables = ('generation_fuel_eia923', 'boiler_fuel_eia923', 'gener`
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_en`
A list of PUDL entity tables.

Type list

`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_epai`
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_use**
A dictionary of EPA IPM tables and associated URLs extensions for downloading that table's data.

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', 'F1**
A dictionary mapping FERC Form 1 DBF files(w / o .DBF file extension) (keys) to database table names (values).

Type dict

pudl.constants.**ferc1_huge_tables = {'f1_footnote_data', 'f1_footnote_tbl', 'f1_note_fin_st**
A set containing large FERC Form 1 tables.

Type set

pudl.constants.**ferc1_power_purchase_type = {'AD': 'adjustment', 'EX': 'electricity_exchang**
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_ferc**
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_2**
A dictionary mapping database table names (keys) to FERC Form 1 DBF files(w / o .DBF file extension) (values).

Type dict

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.**files_dict_epaipm = {'load_curves_epaipm': '*table_2-2_*', 'plant_region_m**
A dictionary of EPA IPM tables and strings that files of those tables contain.

Type dict

`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_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', 'sc`**

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

`pudl.constants.`**`fuel_type_eia860_nuclear_strings = ['nuc', 'nuclear']`**

A list of strings from EIA 860 associated with nuclear power.

Type list

`pudl.constants.`**`fuel_type_eia860_oil_strings = ['dfo', 'jf', 'ker', 'rfo', 'wo', 'woo', 'pet`**

A list of strings from EIA 860 associated with fuel type oil.

Type list

`pudl.constants.`**`fuel_type_eia860_other_strings = ['mwh', 'oth', 'pur', 'wh', 'geo', 'none',`**

A list of strings from EIA 860 associated with fuel type other.

Type list

`pudl.constants.`**`fuel_type_eia860_simple_map = {'coal': ['ant', 'bit', 'cbl', 'lig', 'pc',`**

A dictionary mapping EIA 860 fuel types (keys) to lists of strings associated with that fuel type (values).

Type dict

`pudl.constants.`**`fuel_type_eia860_solar_strings = ['sun', 'solar']`**

A list of strings from EIA 860 associated with solar power.

Type list

`pudl.constants.`**`fuel_type_eia860_waste_strings = ['ab', 'blq', 'bm', 'msb', 'msn', 'obl', 'o`**

A list of strings from EIA 860 associated with fuel type waste.

Type list

`pudl.constants.`**`fuel_type_eia860_wind_strings = ['wnd', 'wind', 'wt']`**

A list of strings from EIA 860 associated with wind power.

Type list

`pudl.constants.`**`fuel_type_eia923 = {'AB': 'Agricultural By-Products', 'ANT': 'Anthracite Coa`**

A dictionary mapping EIA 923 fuel type codes (keys) and fuel type names / descriptions (values).

Type dict

pudl.constants.**fuel_type_eia923_boiler_fuel_coal_strings = ['ant', 'bit', 'lig', 'pc', 'rc**
A list of strings from EIA 923 Boiler Fuel associated with fuel type coal.

Type list

pudl.constants.**fuel_type_eia923_boiler_fuel_gas_strings = ['bfg', 'lfg', 'ng', 'og', 'obg',**
A list of strings from EIA 923 Boiler Fuel associated with fuel type gas.

Type list

pudl.constants.**fuel_type_eia923_boiler_fuel_oil_strings = ['dfo', 'rfo', 'wo', 'jf', 'ker']**
A list of strings from EIA 923 Boiler Fuel associated with fuel type oil.

Type list

pudl.constants.**fuel_type_eia923_boiler_fuel_other_strings = ['oth', 'pur', 'wh']**
A list of strings from EIA 923 Boiler Fuel associated with fuel type other.

Type list

pudl.constants.**fuel_type_eia923_boiler_fuel_simple_map = {'coal': ['ant', 'bit', 'lig', 'p**
A dictionary mapping EIA 923 Boiler Fuel fuel types (keys) to lists of strings associated with that fuel type (values).

Type dict

pudl.constants.**fuel_type_eia923_boiler_fuel_waste_strings = ['ab', 'blq', 'msb', 'msn', 'ob**
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',**
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', 'p**
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', 'jf', '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**
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',**

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**

A dictionary mapping EIA 923 fuel units (keys) to fuel unit descriptions (values).

Type dict

pudl.constants.**glue_pudl_tables = ('plants_eia', 'plants_ferc', 'plants', 'utilities_eia',**

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.**keywords_by_data_source = {'eia860': ['electricity', 'electric', 'boiler',**

A dictionary of datasets (keys) and keywords (values).

Type dict

pudl.constants.**licenses = {'cc-by-4.0': {'name': 'CC-BY-4.0', 'path': 'https://creative**

A dictionary of dictionaries containing license types and their attributes.

Type dict

pudl.constants.**month_dict_eia923 = {1: '_january$', 2: '_february$', 3: '_march$', 4:**

A dictionary mapping column numbers (keys) to months (values).

Type dict

pudl.constants.**need_fix_inting = {'hourly_emissions_epacems': ('facility_id', 'unit_id_epa**

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': 'Flor**

A dictionary mapping NERC Region abbreviations (keys) to NERC Region names (values).

Type dict

pudl.constants.**output_formats = ['sqlite', 'parquet', 'datapkg']**

A list of types of PUDL output formats.

Type list

pudl.constants.**prime_movers = ['steam_turbine', 'gas_turbine', 'hydro', 'internal_combustio**

A list of the types of prime movers

Type list

pudl.constants.**prime_movers_eia923 = {'BA': 'Energy Storage, Battery', 'BT': 'Turbines Used**

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_eia86`**

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.`**`rto_iso = {'CAISO': 'California ISO', 'ERCOT': 'Electric Reliability Counci`**

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': '`**

A dictionary mapping EIA numeric codes (keys) to EIA's internal consolidated NAICS sectors (values).

Type dict

`pudl.constants.`**`state_tz_approx = {'AB': 'America/Edmonton', 'AK': 'US/Alaska', 'AL': 'US/C`**

A dictionary containing US and Canadian state/territory abbreviations (keys) and timezones (values)

Type dict

`pudl.constants.`**`table_map_ferc1_pudl = {'fuel_ferc1': 'f1_fuel', 'plant_in_service_ferc1':`**

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.`**`us_states = {'AK': 'Alaska', 'AL': 'Alabama', 'AR': 'Arkansas', 'AS': 'Ame`**

A dictionary containing US state abbreviations (keys) and names (values)

Type dict

`pudl.constants.`**`working_partitions = {'eia860': {'years': (2004, 2005, 2006, 2007, 2008,`**

A dictionary of data sources (keys) and dictionaries (values) of names of partition type (sub-key) and paritions (sub-value) containing the paritions such as tuples of 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'`**

The location of the xlsx maps within the PUDL package data.

Type string

pudl.dfc module

Implemenation of DataFrameCollection.

Pudl ETL needs to exchange collections of named tables (pandas.DataFrame) between ETL tasks and the volume of data contained in these tables can far exceed the memory of a single machine.

Prefect framework currently caches task results in-memory and this can lead to out of memory problem, especially when dealing with large datasets (e.g. during the full data release). To alleviate this problem, prefect team recommends passing "references" to actual data that is stored separately.

DataFrameCollection does just this. It keeps lightweight references to named data frames and stores the data either locally or on cloud storage (we use pandas.to_pickle method which supports these various storage backends out of the box).

Think of DataFrameCollection as a dict-like structure backed by a disk.

class `pudl.dfc.`**`DataFrameCollection`**(*storage_path: Optional[str] = None*, ***data_frames: Dict[str, pandas.core.frame.DataFrame]*)

Bases: `object`

This class can hold named pandas.DataFrame that are stored on disk or GCS.

This should be used whenever dictionaries of named pandas.DataFrames are passed between prefect tasks. Due to the implicit in-memory caching of task results it is important to keep the in-memory footprint of the exchanged data small.

This wrapper achieves this by maintaining references to tables that themselves are stored on a persistent medium such as local disk of GCS bucket.

This is intended to be used from within prefect flows and new instances can be configured by setting relevant prefect.context variables.

`add_reference`(*name: str*, *table_id: uuid.UUID*)

Adds reference to a named dataframe to this collection.

This assumes that the data is already present on disk.

static **`from_dict`**(*d: Dict[str, pandas.core.frame.DataFrame]*)

Constructs new DataFrameCollection from dataframe dictionary.

`get`(*name: str*) → pandas.core.frame.DataFrame

Returns the content of the named dataframe.

`get_table_names`() → List[str]

Returns sorted list of dataframes that are contained in this collection.

`items`() → Iterator[Tuple[str, pandas.core.frame.DataFrame]]

Iterates over table names and the corresponding pd.DataFrame objects.

`references`() → Iterator[Tuple[str, uuid.UUID]]

Returns a set-like object with (name, table_id) tuples.

`store`(*name: str*, *data: pandas.core.frame.DataFrame*)

Adds named dataframe to collection and stores its contents on disk.

`to_dict`() → Dict[str, pandas.core.frame.DataFrame]

Loads the entire collection to memory as a dictionary.

`union`(**others*)

Returns new DataFrameCollection that is union of self and others.

`update`(*other*)

Adds references to tables from the other DataFrameCollection.

exception `pudl.dfc.`**`TableExists`**

Bases: `Exception`

The table already exists.

Either the table already exists in the DataFrameCollection when it is added or the file containing the serialized form is found on disk.

pudl.etl module

Run the PUDL ETL Pipeline.

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/transfrom/load process for 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.**check_for_bad_tables**(*try_tables*, *dataset*)
Check for bad data tables.

pudl.etl.**check_for_bad_years**(*try_years*, *dataset*)
Check for bad data years.

pudl.etl.**etl**(*datapkg_settings*, *output_dir*, *pudl_settings*, *ds_kwargs*)
Run ETL process for data package specified by datapkg_settings dictionary.

This is the coordinating function for generating all of the CSV's for a data package. For each of the datasets enumerated in the datapkg_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

- **datapkg_settings** (*dict*) – Validated ETL parameters for a single datapackage, originally read in from the PUDL ETL input file.
- **output_dir** (*path-like*) – The individual datapackage directory, which will contain the datapackage.json file and the data directory.
- **pudl_settings** (*dict*) – a dictionary describing paths to various resources and outputs.
- **ds_kwargs** (*dict*) – named-arguments to pass to Datastore constructor when creating new instance. This contains values derived from command-line flags that control how caching layers operate.

Returns The names of the tables included in the output datapackage.

Return type list

pudl.etl.**generate_datapkg_bundle**(*datapkg_bundle_settings*, *pudl_settings*, *datapkg_bundle_name*, *datapkg_bundle_doi=None*, *clobber=False*, *use_local_cache: bool = True*, *gcs_cache_path: Optional[str] = None*)
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

- **datapkg_bundle_settings** (*iterable*) – a list of dictionaries. Each item in the list corresponds to a data package. Each data package's dictionary contains the arguements for its ETL function.
- **pudl_settings** (*dict*) – a dictionary filled with settings that mostly describe paths to various resources and outputs.
- **datapkg_bundle_name** (*str*) – name of directory you want the bundle of data packages to live.
- **clobber** (*bool*) – If True and there is already a directory with data packages with the datapkg_bundle_name, the existing data packages will be deleted and new data packages will be generated in their place.
- **use_local_cache** (*bool*) – controls whether datastore should be using local file cache.
- **gcs_cache_path** (*str*) – controls whether datastore should be using Google Cloud Storage based cache.

Returns A dictionary with datapackage names as the keys, and Python dictionaries representing tabular datapackage resource descriptors as the values, one per datapackage that was generated as part of the bundle.

Return type dict

`pudl.etl.`**`get_flattened_etl_parameters`**(*datapkg_bundle_settings*)

Compile flattened etl parameters.

The datapkg_bundle_settings is a list of dictionaries with the specific etl parameters for each dataset nested inside the dictionary. This function extracts the years, states, tables, etc. from the list datapackage settings and compiles them into one dictionary.

Parameters **datapkg_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 with etl parameter names (keys) (i.e. ferc1_years, eia923_years) and etl parameters (values) (i.e. a list of years for ferc1_years)

Return type dict

`pudl.etl.`**`validate_params`**(*datapkg_bundle_settings*, *pudl_settings*)

Enforce validity of ETL parameters found in datapackage bundle settings.

For each enumerated data package in the datapkg_bundle_settings, this function checks to ensure the input parameters for each of the datasets are consistent with the known input options. Most of those options are enumerated in pudl.constants. For each dataset, the years, states, tables, etc. are checked to ensure that they are valid and present. If parameters are not valid, assertions will be raised.

There is some options that have default options or are hard coded during validation. Tables will typically be defaulted to all of the tables if they aren't set. CEMS is always going to be partitioned by year and state. This means we have functinoally removed the option to not partition or partition another way.

Parameters

- **datapkg_bundle_settings** (*iterable*) – a list of data package parameters, with each element of the list being a dictionary specifying the data to be packaged.
- **pudl_settings** (*dict*) – a dictionary describing paths to various resources and outputs.

Returns

validated list of data package parameters, with each element of the list being a dictionary specitying the data to be packaged.

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 lost of transform type functions in here that help with cleaning and restructing dataframes.

`pudl.helpers.`**`add_fips_ids`**(*df*, *state_col='state'*, *county_col='county'*, *vintage=2015*)
Add State and County FIPS IDs to a dataframe.

`pudl.helpers.`**`clean_eia_counties`**(*df*, *fixes*, *state_col='state'*, *county_col='county'*)
Replace non-standard county names with county nmes from US Census.

`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 `pandas.DataFrame.pipe()` method in a chain.

Parameters

- **df** (*pandas.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 DataFrame containing the cleaned strings.

Return type pandas.DataFrame

`pudl.helpers.`**`cleanstrings_series`**(*col*, *str_map*, *unmapped=None*, *simplify=True*)
Clean up the strings in a single column/Series.

Parameters

- **col** (*pandas.Series*) – A pandas Series, typically a single column of a dataframe, containing the freeform strings that are to be cleaned.

- **str_map** (*dict*) – A dictionary of lists of strings, in which the keys are the simplified canonical strings, witch 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 `pandas.DataFrame`.

Return type pandas.Series

`pudl.helpers.`**`cleanstrings_snake`**(*df*, *cols*)

Clean the strings in a columns in a dataframe with snake case.

Parameters

- **df** (*panda.DataFrame*) – original dataframe.
- **cols** (*list*) – list of columns in *df* to apply snake case to.

`pudl.helpers.`**`convert_cols_dtypes`**(*df*, *data_source*, *name=None*)

Convert the data types for a dataframe.

This function will convert a PUDL dataframe’s columns to the correct data type. It uses a dictionary in constants.py called column_dtypes to assign the right type. Within a given data source (e.g. eia923, ferc1) each column name is assumed to *always* have the same data type whenever it is found.

Boolean type conversions created a special problem, because null values in boolean columns get converted to True (which is bonkers!)… we generally want to preserve the null values and definitely don’t want them to be True, so we are keeping those columns as objects and preforming a simple mask for the boolean columns.

The other exception in here is with the *utility_id_eia* column. It is often an object column of strings. All of the strings are numbers, so it should be possible to convert to `pandas.Int32Dtype()` directly, but it is requiring us to convert to int first. There will probably be other columns that have this problem… and hopefully pandas just enables this direct conversion.

Parameters

- **df** (*pandas.DataFrame*) – dataframe with columns that appear in the PUDL tables.
- **data_source** (*str*) – the name of the datasource (eia, ferc1, etc.)
- **name** (*str*) – name of the table (for logging only!)

Returns a dataframe with columns as specified by the `pudl.constants` `column_dtypes` dictionary.

Return type pandas.DataFrame

`pudl.helpers.`**`convert_dfs_dict_dtypes`**(*dfs_dict*, *data_source*)

Convert the data types of a dictionary of dataframes.

This is a wrapper for `pudl.helpers.convert_cols_dtypes()` which loops over an entire dictionary of dataframes, assuming they are all from the specified data source, and appropriately assigning data types to each column based on the data source specific type map stored in pudl.constants

`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.

If the input `date_col` already exists in the input dataframe, then no conversion is applied, and the original dataframe is returned unchanged. Otherwise the constructed date is placed in that column, and the columns which were used to create the date are dropped.

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.`**`count_records`**(*df*, *cols*, *new_count_col_name*)

Count the number of unique records in group in a dataframe.

Parameters

- **df** (*panda.DataFrame*) – dataframe you would like to groupby and count.
- **cols** (*iterable*) – list of columns to group and count by.
- **new_count_col_name** (*string*) – the name that will be assigned to the column that will contain the count.

Returns dataframe with only the *cols* definted and the *new_count_col_name*.

Return type pandas.DataFrame

`pudl.helpers.`**`download_zip_url`**(*url*, *save_path*, *chunk_size=128*)

Download and save a Zipfile locally.

Useful for acquiring and storing non-PUDL data locally.

Parameters

- **url** (*str*) – The URL from which to download the Zipfile
- **save_path** (*pathlib.Path*) – The location to save the file.
- **chunk_size** (*int*) – Data chunk in bytes to use while downloading.

Returns None

`pudl.helpers.`**`drop_tables`**(*engine*, *clobber=False*)

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 exising SQLite database to be deleted.

Returns None

`pudl.helpers.`**`fillna_w_rolling_avg`**(*df_og*, *group_cols*, *data_col*, *window=12*, ***kwargs*)

Filling NaNs with a rolling average.

Imputes null values from a dataframe on a rolling monthly average. To note, this was designed to work with the PudlTabl object's tables.

Parameters

- **df_og** (*pandas.DataFrame*) – Original dataframe. Must have group_cols column, a data_col column and a 'report_date' column.
- **group_cols** (*iterable*) – a list of columns to groupby.
- **data_col** (*str*) – the name of the data column.
- **window** (*int*) – window from pandas.Series.rolling
- **kwargs** – Additional arguments to pass to `pandas.Series.rolling`.

Returns dataframe with nulls filled in.

Return type pandas.DataFrame

`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.

Currently replaces empty string, single decimal points with no numbers, and any single whitespace character with np.nan.

Parameters **df** (*pandas.DataFrame*) – The DataFrame to clean.

Returns The cleaned DataFrame.

Return type pandas.DataFrame

`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 for use in data packages, 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.

This is an interim solution – now that pandas extension arrays have been implemented, we need to go back through and convert all of these integer columns that contain NA values to Nullable Integer types like Int64.

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)

`pudl.helpers.`**`fix_leading_zero_gen_ids`**(*df*)

Remove leading zeros from EIA generator IDs which are numeric strings.

If the DataFrame contains a column named `generator_id` then that column will be cast to a string, and any all numeric value with leading zeroes will have the leading zeroes removed. This is necessary because in some but not all years of data, some of the generator IDs are treated as integers in the Excel spreadsheets published by EIA, so the same generator may show up with the ID "0001" and "1" in different years.

Alphanumeric generator IDs with leadings zeroes are not affected, as we found no instances in which an alphanumeric generator ID appeared both with and without leading zeroes.

Parameters **df** (*pandas.DataFrame*) – DataFrame, presumably containing a column named generator_id (otherwise no action will be taken.)

Returns pandas.DataFrame

`pudl.helpers.`**`generate_rolling_avg`**(*df*, *group_cols*, *data_col*, *window*, ***kwargs*)

Generate a rolling average.

For a given dataframe with a `report_date` column, generate a monthly rolling average and use this rolling average to impute missing values.

Parameters

- **df** (*pandas.DataFrame*) – Original dataframe. Must have group_cols column, a data_col column and a `report_date` column.
- **group_cols** (*iterable*) – a list of columns to groupby.
- **data_col** (*str*) – the name of the data column.
- **window** (*int*) – window from `pandas.Series.rolling()`.
- **kwargs** – Additional arguments to pass to `pandas.Series.rolling()`.

Returns pandas.DataFrame

`pudl.helpers.`**`get_working_eia_dates`**`()`
Get all working EIA dates as a DatetimeIndex.

`pudl.helpers.`**`is_annual`**`(`*df_year*, *year_col='report_date'*`)`
Determine whether a DataFrame contains consistent annual time-series data.

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. This function attempts to infer the temporal frequency of the dataframe, or if that is impossible, to at least see whether the data would be consistent with annual reporting – e.g. if there is only a single year of data, it should all have the same date, and that date should correspond to January 1st of a given year.

This function is known to be flaky and needs to be re-written to deal with the edge cases better.

Parameters

- **df_year** (*pandas.DataFrame*) – A pandas DataFrame that might contain time-series data at annual resolution.
- **year_col** (*str*) – The column of the DataFrame in which the year is reported.

Returns True if df_year is found to be consistent with continuous annual time resolution, False otherwise.

Return type bool

`pudl.helpers.`**`is_doi`**`(`*doi*`)`
Determine if a string is a valid digital object identifier (DOI).

Function simply checks whether the offered string matches a regular expresssion – it doesn't check whether the DOI is actually registered with the relevant authority.

Parameters **doi** (*str*) – String to validate.

Returns True if doi matches the regex for valid DOIs, False otherwise.

Return type bool

`pudl.helpers.`**`iterate_multivalue_dict`**`(`***kwargs*`)`
Make dicts from dict with main dict key and one value of main dict.

`pudl.helpers.`**`merge_dicts`**`(`*list_of_dicts*`)`
Merge multipe dictionaries together.

Given any number of dicts, shallow copy and merge into a new dict, precedence goes to key value pairs in latter dicts.

Parameters **dict_args** (*list*) – a list of dictionaries.

Returns dict

`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

Raises **ValueError** – if the date or year columns are not found, or if the year column is found to be inconsistent with annual reporting.

Todo: Right mergers will result in null values in the resulting date column. The final output includes the date_col from the date_df and thus if there are any entity records (records being merged on) in the year_df but not in the date_df, a right merge will result in nulls in the date_col. And when we drop the 'year_temp' column, the year from the year_df will be gone. Need to determine how to deal with this. Should we generate a montly record in each year? Should we generate full time serires? Should we restrict right merges in this function?

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.

Todo: 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 automataically 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

`pudl.helpers.oob_to_nan(df, cols, lb=None, ub=None)`

Set non-numeric values and those outside of a given rage to NaN.

Parameters

- **df** (*pandas.DataFrame*) – The dataframe containing values to be altered.
- **cols** (*iterable*) – Labels of the columns whose values are to be changed.
- **lb** – (number): Lower bound, below which values are set to NaN. If None, don't use a lower bound.
- **ub** – (number): Upper bound, below which values are set to NaN. If None, don't use an upper bound.

Returns The altered DataFrame.

Return type pandas.DataFrame

`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

`pudl.helpers.prep_dir(dir_path, clobber=False)`

Create (or delete and recreate) a directory.

Parameters

- **dir_path** (*path-like*) – path to the directory that you are trying to clean and prepare.
- **clobber** (*bool*) – If True and dir_path exists, it will be removed and replaced with a new, empty directory.

Raises **FileExistsError** – if a file or directory already exists at dir_path.

Returns Path to the created directory.

Return type pathlib.Path

`pudl.helpers.simplify_columns(df)`

Simplify column labels for use as snake_case database fields.

All columns will be re-labeled by: * Replacing all non-alphanumeric characters with spaces. * Forcing all letters to be lower case. * Compacting internal whitespace to a single " ". * Stripping leading and trailing whitespace. * Replacing all remaining whitespace with underscores.

Parameters **df** (*pandas.DataFrame*) – The DataFrame to clean.

Returns The cleaned DataFrame.

Return type pandas.DataFrame

Todo: Update docstring.

pudl.helpers.**simplify_strings**(*df*, *columns*)
Simplify the strings contained in a set of dataframe columns.

Performs several operations to simplify strings for comparison and parsing purposes. These include removing Unicode control characters, stripping leading and trailing whitespace, using lowercase characters, and compacting all internal whitespace to a single space.

Leaves null values unaltered. Casts other values with astype(str).

Parameters

- **df** (*pandas.DataFrame*) – DataFrame whose columns are being cleaned up.
- **columns** (*iterable*) – The labels of the string columns to be simplified.

Returns The whole DataFrame that was passed in, with the string columns cleaned up.

Return type pandas.DataFrame

pudl.helpers.**zero_pad_zips**(*zip_series*, *n_digits*)
Retain prefix zeros in zipcodes.

Parameters

- **zip_series** (*pd.Series*) – series containing the zipcode values.
- **n_digits** (*int*) – zipcode length (likely 4 or 5 digits).

Returns a series containing zipcodes with their prefix zeros intact and invalid zipcodes rendered as na.

Return type pandas.Series

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_puc**
EIA923 Boiler Fuel data validation against aggregated data.

pudl.validate.**bf_eia923_coal_ash_content = [{'title': 'Bituminous coal ash content (middle**
Valid coal ash content (%). Based on historical reporting in EIA 923.

pudl.validate.**bf_eia923_coal_heat_content = [{'title': 'Bituminous coal heat content (mid**
Valid coal (bituminous, sub-bituminous, and lignite) heat content values.

pudl.validate.**bf_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.`**`bf_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.`**`bf_eia923_oil_heat_content = [{'title': 'Diesel Fuel Oil heat content (tails`**

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_t`**

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.`**`check_max_rows`**(*df*, *expected_rows=inf*, *margin=0.05*, *df_name=''*)

Validate that a dataframe has less than a maximum number of rows.

`pudl.validate.`**`check_min_rows`**(*df*, *expected_rows=0*, *margin=0.05*, *df_name=''*)

Validate that a dataframe has a certain minimum number of rows.

`pudl.validate.`**`check_unique_rows`**(*df*, *subset=None*, *df_name=''*)

Test whether dataframe has unique records within a subset of columns.

Parameters

- **df** (*pandas.DataFrame*) – DataFrame to check for duplicate records.
- **subset** (*iterable or None*) – Columns to consider in checking for dupes.
- **df_name** (*str*) – Name of the dataframe, to aid in debugging/logging.

Returns

The same DataFrame as was passed in, for use in DataFrame.pipe().

Return type pandas.DataFrame

Raises **ValueError** – If there are duplicate records in the subset of selected columns.

`pudl.validate.`**`frc_eia923_ag_byproduct_heat_content = [{'title': 'Agricultural byproduct he`**

Check for reasonable agricultural byproduct heat contents.

Based on values given in the EIA 923 instructions: https://www.eia.gov/survey/form/eia_923/instructions.pdf

`pudl.validate.`**`frc_eia923_agg = [{'title': 'Coal ash content', 'query': "fuel_type_code_pu`**

EIA923 fuel receipts & costs data validation against aggregated data.

`pudl.validate.`**`frc_eia923_biomass_gas_heat_content = [{'title': 'Other biomass gas heat con`**

Check for reasonable other biomass gas heat contents.

Based on values given in the EIA 923 instructions: https://www.eia.gov/survey/form/eia_923/instructions.pdf

`pudl.validate.`**`frc_eia923_biomass_liquids_heat_content = [{'title': 'Other biomass liquids`**

Check for reasonable other biomass liquids heat contents.

Based on values given in the EIA 923 instructions: https://www.eia.gov/survey/form/eia_923/instructions.pdf

`pudl.validate.`**`frc_eia923_biomass_solids_heat_content = [{'title': 'Other biomass solids he`**

Check for reasonable other biomass solids heat contents.

Based on values given in the EIA 923 instructions: https://www.eia.gov/survey/form/eia_923/instructions.pdf

`pudl.validate.`**`frc_eia923_black_liquor_heat_content = [{'title': 'Black liquor heat content`**
Check for reasonable black liquor heat contents.

Based on values given in the EIA 923 instructions: https://www.eia.gov/survey/form/eia_923/instructions.pdf

`pudl.validate.`**`frc_eia923_blast_furnace_gas_heat_content = [{'title': 'Blast furnace gas he`**
Check for reasonable blast furnace gas heat contents.

Based on values given in the EIA 923 instructions: https://www.eia.gov/survey/form/eia_923/instructions.pdf

`pudl.validate.`**`frc_eia923_coal_ant_heat_content = [{'title': 'Anthracite coal heat content`**
Check for reasonable anthracite coal heat content.

Based on values given in the EIA 923 instructions: https://www.eia.gov/survey/form/eia_923/instructions.pdf

`pudl.validate.`**`frc_eia923_coal_ash_content = [{'title': 'Bituminous coal ash content (middl`**
Valid coal ash content (%). Based on historical reporting in EIA 923.

`pudl.validate.`**`frc_eia923_coal_bit_heat_content = [{'title': 'Bituminous coal heat content`**
Check for reasonable bituminous coal heat content.

Based on values given in the EIA 923 instructions: https://www.eia.gov/survey/form/eia_923/instructions.pdf

`pudl.validate.`**`frc_eia923_coal_cc_heat_content = [{'title': 'Refined coal heat content (ta`**
Check for reasonable refined coal heat content.

Based on values given in the EIA 923 instructions: https://www.eia.gov/survey/form/eia_923/instructions.pdf

`pudl.validate.`**`frc_eia923_coal_lig_heat_content = [{'title': 'Lignite heat content (middle)`**
Check for reasonable lignite coal heat content.

Based on values given in the EIA 923 instructions: https://www.eia.gov/survey/form/eia_923/instructions.pdf

`pudl.validate.`**`frc_eia923_coal_mercury_content = [{'title': 'Coal mercury content (upper ta`**
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 cont`**
Valid coal moisture content, based on historical EIA 923 reporting.

`pudl.validate.`**`frc_eia923_coal_sub_heat_content = [{'title': 'Sub-bituminous coal heat cont`**
Check for reasonable Sub-bituminous coal heat content.

Based on values given in the EIA 923 instructions: https://www.eia.gov/survey/form/eia_923/instructions.pdf

`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_coal_wc_heat_content = [{'title': 'Waste coal heat content (tails`**
Check for reasonable waste coal heat content.

Based on values given in the EIA 923 instructions: https://www.eia.gov/survey/form/eia_923/instructions.pdf

`pudl.validate.`**`frc_eia923_gas_sgc_heat_content = [{'title': 'Coal syngas heat content (tai`**
Check for reasonable coal syngas heat contents.

Based on values given in the EIA 923 instructions: https://www.eia.gov/survey/form/eia_923/instructions.pdf

`pudl.validate.`**`frc_eia923_landfill_gas_heat_content = [{'title': 'Landfill gas heat content`**
Check for reasonable landfill gas heat contents.

Based on values given in the EIA 923 instructions: https://www.eia.gov/survey/form/eia_923/instructions.pdf

`pudl.validate.`**`frc_eia923_muni_solids_heat_content = [{'title': 'Municipal solid waste heat`**

Check for reasonable municipal solid waste heat contents.

Based on values given in the EIA 923 instructions: https://www.eia.gov/survey/form/eia_923/instructions.pdf

`pudl.validate.`**`frc_eia923_natural_gas_heat_content = [{'title': 'Natural gas heat content`**

Check for reasonable natural gas heat contents.

Based on values given in the EIA 923 instructions: https://www.eia.gov/survey/form/eia_923/instructions.pdf

`pudl.validate.`**`frc_eia923_oil_dfo_heat_content = [{'title': 'Diesel Fuel Oil heat content`**

Check for reasonable diesel fuel oil heat contents.

Based on values given in the EIA 923 instructions: https://www.eia.gov/survey/form/eia_923/instructions.pdf

`pudl.validate.`**`frc_eia923_oil_jf_heat_content = [{'title': 'Jet fuel heat content (tails)',`**

Check for reasonable jet fuel heat contents.

Based on values given in the EIA 923 instructions: https://www.eia.gov/survey/form/eia_923/instructions.pdf

`pudl.validate.`**`frc_eia923_oil_ker_heat_content = [{'title': 'Kerosene heat content (tails)`**

Check for reasonable kerosene heat contents.

Based on values given in the EIA 923 instructions: https://www.eia.gov/survey/form/eia_923/instructions.pdf

`pudl.validate.`**`frc_eia923_other_gas_heat_content = [{'title': 'Other gas heat content (tai`**

Check for reasonable other gas heat contents.

Based on values given in the EIA 923 instructions, but with the lower bound set by the expected lower bound of heat content on blast furnace gas (since there were "other" gasses with bounds lower than the expected 0.32 in the data) https://www.eia.gov/survey/form/eia_923/instructions.pdf

`pudl.validate.`**`frc_eia923_petcoke_heat_content = [{'title': 'Petroleum coke heat content (t`**

Check for reasonable petroleum coke heat contents.

Based on values given in the EIA 923 instructions: https://www.eia.gov/survey/form/eia_923/instructions.pdf

`pudl.validate.`**`frc_eia923_petcoke_syngas_heat_content = [{'title': 'Petcoke syngas heat co`**

Check for reasonable petcoke syngas heat contents.

Based on values given in the EIA 923 instructions: https://www.eia.gov/survey/form/eia_923/instructions.pdf

`pudl.validate.`**`frc_eia923_propane_heat_content = [{'title': 'Propane heat content (tails)',`**

Check for reasonable propane heat contents.

Based on values given in the EIA 923 instructions: https://www.eia.gov/survey/form/eia_923/instructions.pdf

`pudl.validate.`**`frc_eia923_rfo_heat_content = [{'title': 'Residual fuel oil heat content (ta`**

Check for reasonable residual fuel oil heat contents.

Based on values given in the EIA 923 instructions: https://www.eia.gov/survey/form/eia_923/instructions.pdf

`pudl.validate.`**`frc_eia923_self = [{'title': 'Bituminous coal ash content', 'query': "energ`**

EIA923 fuel receipts & costs data validation against itself.

`pudl.validate.`**`frc_eia923_sludge_heat_content = [{'title': 'Sludge waste heat content (tai`**

Check for reasonable sludget waste heat contents.

Based on values given in the EIA 923 instructions: https://www.eia.gov/survey/form/eia_923/instructions.pdf

`pudl.validate.`**`frc_eia923_waste_oil_heat_content = [{'title': 'Waste oil heat content (tai`**

Check for reasonable waste oil heat contents.

Based on values given in the EIA 923 instructions: https://www.eia.gov/survey/form/eia_923/instructions.pdf

`pudl.validate.`**`frc_eia923_wood_liquids_heat_content = [{'title': 'Wood waste liquids heat c`**

Check for reasonable wood waste liquids heat contents.

Based on values given in the EIA 923 instructions: https://www.eia.gov/survey/form/eia_923/instructions.pdf

`pudl.validate.`**`frc_eia923_wood_solids_heat_content = [{'title': 'Wood solids heat content`**

Check for reasonable wood solids heat contents.

Based on values given in the EIA 923 instructions: https://www.eia.gov/survey/form/eia_923/instructions.pdf

`pudl.validate.`**`gf_eia923_agg = [{'title': 'Coal heat content', 'query': "fuel_type_code_pu`**

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.

`pudl.validate.`**`gf_eia923_gas_heat_content = [{'title': 'All gas heat content (middle)', 'qu`**

Valid natural gas heat content values.

Focuses on natural gas proper. Lower bound excludes other types of gaseous fuels intentionally.

`pudl.validate.`**`gf_eia923_oil_heat_content = [{'title': 'Diesel Fuel Oil heat content (tails`**

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`** (`str`) – Label of the column containing the data of interest.
- **`weight_col`** (`str`) – 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)', 'que`**

Static constraints on coal fired generator capacity factors.

`pudl.validate.`**`mcoe_coal_heat_rate = [{'title': 'Coal Unit Heat Rates (middle)', 'query':`**

Static constraints on coal fired generator heat rates.

`pudl.validate.`**`mcoe_fuel_cost_per_mmbtu = [{'title': 'Coal Fuel Costs (middle)', 'query':`**

Static constraints on fuel costs per mmbtu of fuel consumed.

`pudl.validate.`**`mcoe_fuel_cost_per_mwh = [{'title': 'Coal Fuel Costs (middle)', 'query': ":`**
Static constraints on fuel costs per MWh net generation.

`pudl.validate.`**`mcoe_gas_capacity_factor = [{'title': 'Natural Gas Capacity Factor (middle,`**
Static constraints on natural gas generator capacity factors.

`pudl.validate.`**`mcoe_gas_heat_rate = [{'title': 'Natural Gas Unit Heat Rates (middle, 2015+)`**
Static constraints on gas fired generator heat rates.

`pudl.validate.`**`no_null_cols`**(*df*, *cols='all'*, *df_name=''*)
Check that a dataframe has no all-NaN columns.

Occasionally in the concatenation / merging of dataframes we get a label wrong, and it results in a fully NaN column… which should probably never actually happen. This is a quick verification.

Parameters

- **df** (*pandas.DataFrame*) – DataFrame to check for null columns.
- **cols** (*iterable or "all"*) – The labels of columns to check for all-null values. If "all" check all columns.
- **df_name** (*str*) – Name of the dataframe, to aid in debugging/logging.

Returns

The same DataFrame as was passed in, for use in DataFrame.pipe().

Return type pandas.DataFrame

Raises **ValueError** – If any completely NaN / Null valued columns are found.

`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

INDEX

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

X

Y

Z