Table of Contents

[**A.** **Main Scripts** 3](#_Toc206060679)

[1. download\_mastr.py 3](#_Toc206060680)

[2. extract\_zip.py 3](#_Toc206060681)

[3. validate\_xml.py 3](#_Toc206060682)

[4. xml\_to\_json.py 3](#_Toc206060683)

[5. valid\_json.py 3](#_Toc206060684)

[6. list\_states.py 3](#_Toc206060685)

[7. list\_energy\_types.py 4](#_Toc206060686)

[8. list\_years.py 4](#_Toc206060687)

[9. filter\_json\_by\_energy\_code.py 4](#_Toc206060688)

[10. filter\_json\_by\_state\_bundesland.py 4](#_Toc206060689)

[11. filter\_json\_by\_state\_gemeindeschluessel.py 4](#_Toc206060690)

[12. filter\_json\_by\_installation\_year.py 5](#_Toc206060691)

[13. json\_to\_geojson\_batch.py 5](#_Toc206060692)

[14. generate\_geojson\_by\_state\_gemeindeschluessel.py 5](#_Toc206060693)

[15. generate\_geojson\_by\_state\_polygons.py 5](#_Toc206060694)

[16. generate\_geojson\_by\_installation\_year.py 6](#_Toc206060695)

[17. generate\_geojson\_by\_state\_polygons\_yearly 6](#_Toc206060696)

[**B.** **Merged Scripts for Automation** 6](#_Toc206060697)

[1. merged.py 6](#_Toc206060698)

[2. merged\_path.json 7](#_Toc206060699)

[**C.** **QGIS Python Console Scripts** 7](#_Toc206060700)

[1. 1\_ load\_osm.py 7](#_Toc206060701)

[2. 2\_ crs\_zoom.py 7](#_Toc206060702)

[3. 3\_ load\_gadm.py 7](#_Toc206060703)

[4. 4\_ load\_allGermany\_geojson.py 7](#_Toc206060704)

[5. 5\_ load\_by\_state\_gemeindeschluessel.py 8](#_Toc206060705)

[6. 6\_ load\_by\_state\_polygon.py 8](#_Toc206060706)

[7. 7\_ load\_by\_year.py 8](#_Toc206060707)

[8. 8\_ load\_states\_by\_year.py 8](#_Toc206060708)

[9. 9\_ legend.py 9](#_Toc206060709)

[10. 10\_ states\_energytype\_counts\_yearly.py 9](#_Toc206060710)

[11. 11\_ states\_energytype\_counts\_pieChart.py 9](#_Toc206060711)

[12. 12\_ states\_plants\_yearly.py 9](#_Toc206060712)

[13. 13\_ states\_yearly\_power\_trend.py 10](#_Toc206060713)

[**D.** **Unit Tests** 10](#_Toc206060714)

[1. conftest.py 10](#_Toc206060715)

[2. test\_download\_mastr.py 10](#_Toc206060716)

[3. test\_extract\_zip.py 10](#_Toc206060717)

[4. test\_validate\_xml.py 11](#_Toc206060718)

[5. test\_xml\_to\_json.py 11](#_Toc206060719)

[6. test\_valid\_json.py 11](#_Toc206060720)

[7. test\_list\_states.py 11](#_Toc206060721)

[8. test\_list\_energy\_types.py 12](#_Toc206060722)

[9. test\_list\_years.py 12](#_Toc206060723)

[10. test\_filter\_json\_by\_energy\_code.py 12](#_Toc206060724)

[11. test\_filter\_json\_by\_state\_bundesland.py 12](#_Toc206060725)

[12. test\_filter\_json\_by\_state\_gemeindeschluessel.py 13](#_Toc206060726)

[13. test\_filter\_json\_by\_installation\_year.py 13](#_Toc206060727)

[14. test\_json\_to\_geojson\_batch.py 13](#_Toc206060728)

[15. test\_generate\_geojson\_by\_state\_gemeindeschluessel.py 13](#_Toc206060729)

[16. test\_generate\_geojson\_by\_state\_polygons.py 14](#_Toc206060730)

[17. test\_generate\_geojson\_by\_installation\_year.py 14](#_Toc206060731)

[18. test\_generate\_geojson\_by\_state\_polygons\_yearly 15](#_Toc206060732)

# **Main Scripts**

## download\_mastr.py

This script is used to automatically download the MaStR (Marktstammdatenregister) dataset in ZIP format from a specified URL.

A destination folder for storing the raw files is created if it does not already exist.

The file is retrieved using the requests library in streamed chunks and saved locally.

Upon successful completion, the full path to the downloaded file is returned.

## extract\_zip.py

This script is used to extract all ZIP files located in a specified input directory.

An output directory is created if it does not already exist.

Each ZIP file is decompressed into its own subfolder named after the file (without the extension).

A confirmation message is displayed upon successful extraction of each file.

## validate\_xml.py

This script is used to verify the validity of XML files within a given directory.

Each file is parsed to detect potential formatting errors.

Files that are successfully validated are copied to a designated output directory.

A summary of the total number of scanned files, the count of valid files, and the count of invalid files is displayed upon completion.

## xml\_to\_json.py

This script is used to convert XML files into JSON format.

Each XML file is parsed, and its elements are transformed into key–value pairs stored in a list.

The resulting list is written to a JSON file with proper indentation and UTF-8 encoding.

All valid XML files from the input directory are processed in batch, and the generated JSON files are saved in a specified output directory.

## valid\_json.py

This script is used to filter JSON files and retain only entries containing all required keys with non-empty values.

Each JSON file in the input directory is read and validated against a predefined set of keys.

Only valid entries are written to new JSON files in the output directory.

A summary is displayed showing the total number of processed files and the total count of valid entries extracted.

## list\_states.py

This script is used to identify all unique state codes (Bundesland) present in a collection of JSON files.

Each file in the specified directory is scanned, and the values of the Bundesland field are collected.

The distinct codes are displayed in sorted order once the scan is complete.

## list\_energy\_types.py

This script is used to collect all unique energy type codes (Energieträger) from a set of JSON files.

Each file in the specified directory is scanned, and the values of the designated key are stored in a set.

The distinct codes are then displayed in sorted order upon completion of the scan.

## list\_years.py

This script is used to identify and count installation years from a collection of JSON files.

The year is extracted from the specified date field by taking the first four characters of the date string.

Each occurrence is counted, and the results are displayed in ascending order of year along with the corresponding entry counts.

## filter\_json\_by\_energy\_code.py

This script is used to filter JSON files based on specified energy type codes.

Each file in the input directory is read, and entries matching the given codes in the designated key are extracted.

Filtered entries are saved into separate subdirectories named after each energy code within the output directory.

A count of saved entries is displayed for each processed file and code combination.

## filter\_json\_by\_state\_bundesland.py

This script is used to filter JSON files according to specified state codes in the Bundesland field.

Each JSON file in the input directory is read, and entries matching each state code are extracted.

The filtered entries are saved into separate subdirectories named after the corresponding state code within the output directory.

A count of saved entries is displayed for each processed file and state code combination.

## filter\_json\_by\_state\_gemeindeschluessel.py

This script is used to group and filter JSON entries based on the first two digits of their Gemeindeschluessel value, which correspond to state codes.

Each JSON file in the input directory is scanned, and entries are categorized into state-specific collections according to their prefix.

The filtered entries for each state are saved into separate subdirectories named after the prefix within the output directory.

A count of saved entries is displayed for each file and state prefix combination.

## filter\_json\_by\_installation\_year.py

This script is used to filter JSON entries according to their installation year, as specified in the Inbetriebnahmedatum field.

Years are extracted from date strings and validated to fall between 1900 and 2025.

Entries are grouped by year, and each group is saved into a separate subdirectory named after the year within the output directory.

A count of saved entries is displayed for each file and year combination.

## json\_to\_geojson\_batch.py

This script is used to convert multiple JSON files into a single GeoJSON file containing point features.

Longitude and latitude values are extracted from each entry, validated, and used to create point geometries.

Entries without valid coordinates are skipped.

All valid features are combined into a FeatureCollection and saved to the specified GeoJSON output path.

A summary of processed files and total features written is displayed upon completion.

## generate\_geojson\_by\_state\_gemeindeschluessel.py

This script is used to generate separate GeoJSON files for each German state, based on the first two digits of the Gemeindeschluessel field.

Longitude and latitude values are extracted and validated to create point geometries, while all other attributes are stored as properties.

Entries are grouped by their state prefix, and each group is saved as a FeatureCollection in a state-specific GeoJSON file.

A summary of processed JSON files and the number of features saved for each state is displayed upon completion.

## generate\_geojson\_by\_state\_polygons.py

This script is used to assign power plant locations to German states based on polygon boundaries and generate state-specific GeoJSON files.

State boundaries are loaded from a predefined polygon dataset and standardized as MultiPolygon geometries.

Each JSON entry is converted into a point feature using its latitude and longitude values.

Points are checked for containment within state polygons, and matched entries are grouped by state name.

Each group is saved as a FeatureCollection in a separate GeoJSON file, while unmatched entries are counted and reported.

## generate\_geojson\_by\_installation\_year.py

This script is used to generate GeoJSON files grouped by installation year of power plants.

Years are extracted from the Inbetriebnahmedatum field and validated to be within the range 1900–2025.

Each valid entry is converted into a point feature using its coordinates, and grouped according to its installation year.

A separate FeatureCollection GeoJSON file is created for each year containing the relevant features.

Counts of processed files, valid entries, and skipped entries are displayed upon completion.

## generate\_geojson\_by\_state\_polygons\_yearly

This script is used to assign power plant locations to German states based on polygon boundaries and further separate them by installation year.

State boundaries are loaded from a polygon dataset and standardized as MultiPolygon geometries.

Each JSON entry is converted into a point feature from its coordinates and matched to the state polygon that contains it.

Matched features are grouped first by state and then by installation year, with each group saved as a separate GeoJSON file in a state-specific subdirectory.

Counts of processed files, matched entries, and unmatched entries are displayed upon completion.

# **Merged Scripts for Automation**

## merged.py

This Python script functions as a complete end-to-end ETL (Extract–Transform–Load) pipeline for MaStR data. It sequentially automates the following steps: downloading the official ZIP dataset, extracting its contents, validating XML files, converting them into JSON, filtering JSON entries based on required keys, generating a consolidated GeoJSON of all valid entries, and optionally creating per-state GeoJSON files using polygon boundaries.

- download\_mastr.py

- extract\_zip.py

- validate\_xml.py

- xml\_to\_json.py

- valid\_json.py

- json\_to\_geojson\_batch.py

- generate\_geojson\_by\_state\_polygons.py

The script reads all paths, URLs, and settings from the accompanying merged\_path.json configuration file, ensuring flexibility and reusability.

## merged\_path.json

This configuration file defines all runtime parameters for merged.py. It includes the MaStR download URL, folder paths for raw, extracted, validated, and processed data, output locations for generated GeoJSON files, polygon file location for state-based processing, and a list of required keys for filtering valid JSON entries. By modifying this file, the pipeline can be adapted to different data sources, directory structures, and validation requirements without changing the main script logic.

# **QGIS Python Console Scripts**

## 1\_ load\_osm.py

This script is used to load an OpenStreetMap basemap as an XYZ tile layer into QGIS.

The layer is created using the OSM tile URL and added to the current QGIS project if successfully validated.

A confirmation message is displayed upon successful addition, or an error message is shown if loading fails.

## 2\_ crs\_zoom.py

This script is used to zoom the QGIS map canvas to the geographic extent of Germany.

The extent is first defined in EPSG:4326 (latitude/longitude) coordinates and then transformed into the current project CRS.

The transformed bounding box is applied to the map canvas, which is refreshed to display the updated view.

## 3\_ load\_gadm.py

This script is used to load multiple GADM administrative boundary layers for Germany into QGIS.

Each JSON file is loaded as a vector layer, added to the project, and stored in a reference dictionary if successfully validated.

Opacity is set to 50% for all loaded layers, and only the state boundary layer (gadm41\_DEU\_1) is kept visible.

State names from this layer are labeled using the NAME\_1 field, with customized font, size, and color settings.

## 4\_ load\_allGermany\_geojson.py

This script is used to load and style the all\_germany.geojson layer in QGIS based on energy type codes.

Each energy type is assigned a specific color, label, and symbol size that scales logarithmically with Bruttoleistung values.

The outline color is dynamically set to green for remotely controllable plants and black otherwise.

A rule-based renderer is applied to categorize and symbolize the features, and the styled layer is added to the QGIS project.

## 5\_ load\_by\_state\_gemeindeschluessel.py

This script is used to load and style multiple state-specific GeoJSON layers categorized by Gemeindeschluessel prefixes.

An existing QGIS group with the designated name is removed if present, and a new group is created to store the layers.

Each GeoJSON file is styled using a rule-based renderer, where energy type codes determine symbol color, label text, and size scaling based on Bruttoleistung.

Outline color is dynamically set to green for remotely controllable plants and black otherwise.

The styled layers are added to the newly created group within the QGIS project.

## 6\_ load\_by\_state\_polygon.py

This script is used to load and style multiple state-specific GeoJSON layers where state assignment is determined by polygon boundaries.

An existing QGIS group with the designated name is replaced if present, and a new group is created for storing the layers.

Each GeoJSON file is styled using a rule-based renderer, where energy type codes determine symbol color, label text, and size scaling according to Bruttoleistung.

Outline colors are dynamically set to green for remotely controllable plants and black otherwise.

The styled layers are inserted into the created group within the QGIS project.

## 7\_ load\_by\_year.py

This script is used to load and style GeoJSON layers representing power plants categorized by installation year.

An existing QGIS group with the designated name is replaced if present, and a new group is created to store the layers.

Each file is styled using a rule-based renderer where energy type codes define the symbol color, label text, and size scaling based on Bruttoleistung.

Outline colors are set dynamically to green for remotely controllable plants and black otherwise.

The styled layers are added to the created group within the QGIS project.

## 8\_ load\_states\_by\_year.py

This script is used to load and style GeoJSON layers representing power plants grouped by both state and installation year.

An existing QGIS group with the designated name is removed if present, and a new main group is created.

For each state subdirectory, yearly GeoJSON files are loaded into subgroups, styled using a rule-based renderer, and categorized by energy type code.

Symbol color, label text, and size scaling are determined by the energy type and Bruttoleistung values, while outline color is set to green for remotely controllable plants and black otherwise.

Layers are initially set to be invisible and organized under their respective state and year groups within the QGIS project.

## 9\_ legend.py

This script is used to create a custom legend illustrating the symbology used for energy plant visualization.

Colored markers are plotted alongside descriptive labels to represent different energy types.

Additional notes explain that symbol size is proportional to power capacity and that outline color indicates remote controllability.

The layout is manually adjusted to ensure all labels and markers are clearly visible, and the legend is displayed using Matplotlib.

## 10\_ states\_energytype\_counts\_yearly.py

This script is used to visualize yearly counts of power plants by energy type for each German state.

GeoJSON files from state-specific yearly directories are read, and energy type codes are mapped to descriptive labels.

Counts are aggregated in a nested structure organized by state, year, and energy type.

For each state, a stacked bar chart is created showing the yearly distribution of energy types, with colors assigned according to a predefined scheme.

The charts are displayed in a tabbed QGIS dialog, allowing interactive navigation with zoom and pan tools.

## 11\_ states\_energytype\_counts\_pieChart.py

This script is used to visualize the distribution of power plants by energy type for each German state as pie charts.

Energy type codes from state-specific yearly GeoJSON files are converted to descriptive labels and counted.

Categories representing less than 1% of the total for a state are grouped under the "Other" category.

Pie charts are styled with predefined colors, displayed in separate tabs for each state, and include legends indicating the energy type categories.

The charts are presented in a floating QGIS dialog with navigation and zoom tools.

## 12\_ states\_plants\_yearly.py

This script is used to display the yearly number of power plants commissioned in each German state.

State-specific yearly GeoJSON files are read, and the number of features in each file is counted as the number of plants for that year.

For each state, a bar chart is created showing annual counts, with values labeled above the bars.

The charts are presented in a tabbed QGIS dialog, allowing navigation between states, and are styled with background colors and grid lines for clarity.

## 13\_ states\_yearly\_power\_trend.py

This script is used to display yearly trends of total installed power (Bruttoleistung) for each German state.

State-specific yearly GeoJSON files are read, and power values are extracted, converted to kilowatts, and summed by year.

For each state, a line chart is created showing the evolution of total installed power over time, with data labels added above each point.

The charts are presented in a tabbed QGIS dialog, allowing navigation between states, and styled with background colors, grid lines, and rotated year labels for clarity.

# **Unit Tests**

## conftest.py

This configuration script is used to set up pytest for unit testing of the project’s scripts.

The project’s scripts directory is added to sys.path to allow direct importing of modules during testing.

Reusable fixtures are provided, including temporary directories for downloads, a sample MaStR export URL, and a factory fixture for creating ZIP files with specified contents.

These fixtures facilitate consistent and isolated test environments.

## test\_download\_mastr.py

This script is used to unit test the download\_file function from download\_mastr.py using pytest.

The requests.get method is mocked to avoid real network calls, returning controlled fake responses instead.

Tests verify that destination folders are created when missing, existing folders are not recreated, and streamed data is written correctly to disk.

Additional checks confirm that the correct chunk size is passed to iter\_content, that HTTP errors are propagated, and that the returned file path matches expectations.

## test\_extract\_zip.py

This script is used to unit test the extract\_all\_zips function from extract\_zip.py using pytest.

Tests verify that the function handles various scenarios, including missing input directories, absence of ZIP files, creation of output directories, and proper extraction of single or multiple ZIP archives.

Additional checks confirm that non-ZIP files are ignored, extraction preserves folder structures and file contents, and corrupted ZIP files raise a zipfile.BadZipFile exception.

Console output is also examined to ensure that progress and completion messages are printed as expected.

## test\_validate\_xml.py

This script is used to unit test the XML validation and copying functions from validate\_xml.py using pytest.

Tests confirm that is\_valid\_xml correctly identifies well-formed and malformed XML files, printing appropriate scan and error messages.

The validate\_and\_copy\_xmls function is verified to copy only valid XML files, ignore non-XML files, and produce accurate counts of valid and invalid files.

Behavior is also checked when the input directory is empty, ensuring that no files are copied and that zero counts are reported.

## test\_xml\_to\_json.py

This script is used to unit test the XML-to-JSON conversion functions from xml\_to\_json.py using pytest.

The xml\_file\_to\_json function is tested to ensure valid XML files are correctly converted to JSON format, while invalid XML files are skipped with appropriate warning messages.

The batch\_convert\_xml\_to\_json function is verified to process only .xml files in a directory, skip non-XML files, and produce correctly structured JSON outputs for each valid file.

Output messages and generated file contents are checked to confirm accurate conversion results.

## test\_valid\_json.py

This script is used to unit test the JSON validation and filtering logic from valid\_json.py using pytest.

The is\_valid function is tested with multiple cases to confirm that only entries containing all required keys with non-empty values are considered valid.

An integration-style test simulates file processing by creating temporary JSON files containing valid, invalid, and malformed data.

The test verifies that only valid entries are saved to the output directory, malformed JSON files are skipped with warnings, and console output correctly reflects processing results and summary counts.

## test\_list\_states.py

This script is used to unit test the state code listing functionality from list\_states.py using pytest.

Tests verify that all JSON files in a directory are scanned, unique Bundesland codes are identified, duplicates are removed, and codes are displayed in sorted order.

Additional checks confirm that non-JSON files are ignored, empty or missing Bundesland values are excluded, and invalid JSON files produce appropriate warning messages.

Behavior is also tested for an empty folder, ensuring that no codes are listed while still displaying the summary message.

## test\_list\_energy\_types.py

This script is used to unit test the energy type code listing functionality from list\_energy\_types.py using pytest.

Tests verify that all JSON files in a directory are scanned, unique energy type codes are identified, duplicates are removed, and results are displayed in sorted order.

Functionality is also checked for using a custom key instead of the default Energietraeger field, ensuring that values are correctly extracted and listed regardless of key name.

Additional tests confirm that non-JSON files are ignored, empty values are excluded, and invalid JSON files produce appropriate warning messages without interrupting execution.

## test\_list\_years.py

This script is used to unit test the installation year listing functionality from list\_years.py using pytest.

Tests confirm that JSON files are scanned, years are extracted from the Inbetriebnahmedatum field, counts per year are tallied, and results are displayed in sorted order.

Functionality is also validated for using a custom date key, ensuring correct extraction and counting of years.

Additional checks verify that non-JSON files are ignored, empty values are excluded, invalid JSON files produce warning messages, and the extract year helper function correctly handles valid dates, non-date strings, empty strings, and None values.

## test\_filter\_json\_by\_energy\_code.py

This script is used to unit test the energy code–based filtering functionality from filter\_json\_by\_energy\_code.py using pytest.

Tests confirm that the output base directory and per-code subdirectories are created, and that filtered files are written with the correct filenames and content.

Additional checks verify that non-JSON files are ignored, corrupted JSON files produce warnings without interrupting processing, and no output files are created when no entries match the specified codes.

Functionality for using a custom key instead of Energietraeger is also validated, ensuring that filtering works regardless of the key name.

## test\_filter\_json\_by\_state\_bundesland.py

This script is used to unit test the state code–based filtering functionality from filter\_json\_by\_state\_bundesland.py using pytest.

Tests confirm that the output base directory and per-code subdirectories are created, and that filtered JSON files are written with correct filenames and content.

Checks ensure that non-JSON files are ignored, corrupted JSON files produce warnings without halting execution, and subdirectories are pre-created even when no entries match the target codes.

Functionality for using a custom key instead of Bundesland is validated, along with verification that progress and save messages are printed during processing.

## test\_filter\_json\_by\_state\_gemeindeschluessel.py

This script is used to unit test the filtering of JSON entries by state prefix derived from the Gemeindeschluessel field in filter\_json\_by\_state\_gemeindeschluessel.py using pytest.

The extract\_state\_prefix function is tested for multiple edge cases and valid inputs to ensure correct prefix extraction.

Integration tests verify that the base output directory is created, per-prefix subdirectories are generated only for prefixes with matching entries, filenames are preserved, and contents are correctly filtered.

Additional checks confirm that non-JSON files are ignored, corrupted JSON files trigger warnings without halting execution, and no subdirectories are created when no valid prefixes are found.

Console output is inspected to ensure that progress and save messages are displayed appropriately.

## test\_filter\_json\_by\_installation\_year.py

This script is used to unit test the year-based filtering functionality from filter\_json\_by\_installation\_year.py using pytest.

The extract\_year function is tested for various valid and invalid formats to ensure correct extraction of four-digit years from date strings.

Integration tests verify that year folders for 1900–2025 are created, matching entries are written to the correct subdirectories, and non-matching years are excluded.

Checks confirm that non-JSON files are ignored, corrupted JSON files trigger warnings without halting execution, and no output files are produced for years with zero matches.

Functionality for using a custom year key instead of Inbetriebnahmedatum is validated, ensuring proper filtering and grouping regardless of the field name.

## test\_json\_to\_geojson\_batch.py

This script is used to unit test the JSON-to-GeoJSON batch conversion functionality from json\_to\_geojson\_batch.py using pytest.

The create\_feature function is tested to ensure that valid coordinate values produce correct point geometries with properties, and that invalid or missing coordinates result in None.

The convert\_all\_json\_to\_geojson function is verified to scan JSON files in a directory, skip invalid files, convert only valid entries to GeoJSON features, and write them to a FeatureCollection file.

Tests confirm that output files are created, the correct number of features is included, and console messages report scanning, warnings, and completion with summary counts.

Functionality for handling empty input directories is also validated, ensuring that an empty FeatureCollection is produced with an appropriate message.

## test\_generate\_geojson\_by\_state\_gemeindeschluessel.py

This script is used to unit test the functionality of generate\_geojson\_by\_state\_gemeindeschluessel.py using pytest.

The create\_feature function is tested to ensure that coordinates with comma decimal separators are converted to floats, out-of-bounds or non-numeric values are rejected, and coordinate fields are excluded from properties.

Integration tests verify that JSON files are scanned recursively, non-JSON files are ignored, corrupted JSON files trigger warnings, and entries are grouped by the first two digits of Gemeindeschluessel.

Output GeoJSON files are confirmed to use mapped state names when available, fallback names otherwise, and to contain only valid coordinate entries.

Additional checks validate that progress, save, and summary messages are printed, and that no output files are created when there are no valid entries.

## test\_generate\_geojson\_by\_state\_polygons.py

This script is used to unit test the functionality of generate\_geojson\_by\_state\_polygons.py using pytest.

The create\_feature function is tested to ensure that coordinates with comma decimal separators are correctly converted to floats, out-of-bounds or non-numeric values are rejected, and coordinate keys are excluded from feature properties.

Integration tests for convert\_jsons verify that JSON files are scanned recursively, non-JSON files are ignored, corrupted JSON files produce warnings, and valid entries are matched to states using monkeypatched polygon geometries.

Matched entries are grouped by state name, saved as <state>.geojson files containing only valid coordinates, and unmatched entries are logged.

Additional tests confirm that when no matches are found or polygons are empty, no output files are created while progress messages are still displayed.

## test\_generate\_geojson\_by\_installation\_year.py

This script is used to unit test the functionality of generate\_geojson\_by\_installation\_year.py using pytest.

The extract\_year function is tested to confirm correct extraction of four-digit years from various valid date formats and rejection of invalid, out-of-range, non-string, or incorrectly formatted values.

The create\_feature function is validated to ensure that coordinates with comma decimal separators are converted to floats, out-of-bounds or non-numeric values are rejected, and coordinate fields are excluded from feature properties.

Integration tests for convert\_jsons\_by\_year verify that JSON files in nested directories are scanned, non-JSON files are ignored, corrupted JSON files produce warnings, and valid entries are grouped by installation year.

Each group is saved as a <YEAR>.geojson file containing only valid coordinate entries, with properties preserved except for coordinate fields.

Additional tests confirm that progress, save, and summary messages are printed and that no output files are created when there are no valid entries.

## test\_generate\_geojson\_by\_state\_polygons\_yearly

This script is used to unit test the functionality of generate\_geojson\_by\_state\_polygons\_yearly.py using pytest.

The create\_feature function is tested to confirm that coordinates with comma decimal separators are converted to floats, out-of-bounds or non-numeric values are rejected, and coordinate fields are excluded from properties.

Integration tests for convert\_jsons verify that JSON files are scanned recursively, non-JSON files are ignored, corrupted JSON files produce warnings, and valid entries are matched to states using monkeypatched polygon geometries.

Matched entries are grouped first by state name and then by the first four characters of the Inbetriebnahmedatum field, with “unknown” assigned when missing or too short.

Each group is saved as <output>/<StateName>/<YEAR>.geojson containing only valid coordinate entries.

Additional tests confirm that unmatched entries are logged, progress and save messages are printed, and no GeoJSON files are created when there are no matched entries.