ixmp

ixmp Developers

Nov 21, 2023

GETTING STARTED

The *ix modeling platform* (ixmp) is a data warehouse for high-powered numerical scenario analysis. It is designed to provide an effective framework for integrated and cross-cutting analysis (hence the abbreviation *ix*).

Fig. 1: Key features of the *ix modeling platform* [2]

The platform allows an efficient workflow between original input data sources and implementations of mathematical models, via application programming interfaces (API) with the scientific programming languages Python and R. The platform also includes an API with the mathematical programming software GAMS.

For the scientific reference, see Huppmann et al. (2019) [2].

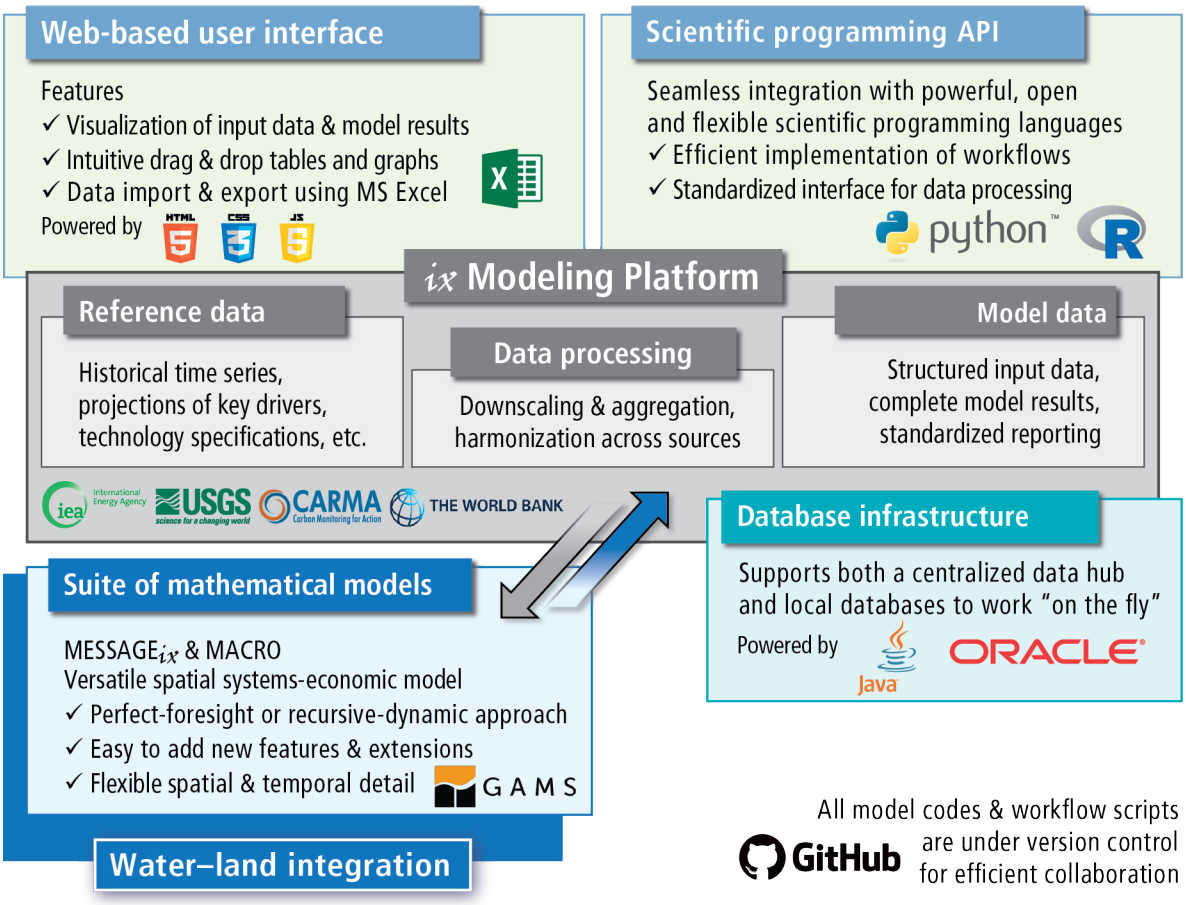

Fig. 2: Components and their interlinkages in the *ix modeling platform* (source [2]): web-based user interface, scientific programming interface, modeling platform, database backend, implementation of the MESSAGE*ix* mathematical model formulation

CHAPTER

ONE

GETTING STARTED

1.1 Installation

Most users will have *ixmp* installed automatically when installing MESSAGEix. The sections below cover other use cases.

Ensure you have first read the prerequisites for understanding and using MESSAGE*ix*. These include specific points of knowledge that are necessary to understand these instructions and choose among different installation options.

To use *ixmp* from R, see Install R and reticulate in the MESSAGE*ix* documentation.

Use cases for installing ixmp directly include:

- Installing *ixmp* to be used alone (i.e., with models or frameworks other than MESSAGE*ix*). Follow the sections:
  - *Install system dependencies*, then
  - *Using Anaconda*.
- Installing *ixmp* from source, for development purposes, e.g. to be used with a source install of `message_ix`. Follow the sections:
  - *Install system dependencies*, then
  - *From source*.

Contents:

1.1.1 Install system dependencies

Python

Python version 3.8 or later is required.

GAMS (required)

ixmp requires GAMS.

1. Download GAMS for your operating system; either the latest version or, for users not familiar with GAMS licenses, version 29 (see note below).
2. Run the installer.
3. Ensure that the `PATH` environment variable on your system includes the path to the GAMS program:
   - on Windows, in the GAMS installer…
     - Check the box labeled "Use advanced installation mode."
     - Check the box labeled "Add GAMS directory to PATH environment variable" on the Advanced Options page.
   - on other platforms (macOS or Linux), add the following line to a file such as `~/.bash_profile` (macOS), `~/.bashrc`, or `~/.profile`:

     ```
     export PATH=$PATH:/path/to/gams-directory-with-gams-binary
     ```

Note: `message_ix` requires GAMS version 24.8; *ixmp* has no minimum requirement *per se*. The latest version is recommended.

GAMS is proprietary software and requires a license to solve optimization problems. To run both the *ixmp* and `message_ix` tutorials and test suites, a "free demonstration" license is required; the free license is suitable for these small models. Versions of GAMS up to version 29 include such a license with the installer; since version 30, the free demo license is no longer included, but may be requested via the GAMS website.

Note: If you only have a license for an older version of GAMS, install both the older and the latest versions.

Graphviz (optional)

`ixmp.Reporter.visualize` uses Graphviz, a program for graph visualization. Installing ixmp causes the graphviz Python package to be installed. If you want to use `visualize()` or run the test suite, the Graphviz program itself must also be installed; otherwise it is **optional**.

If you install *ixmp* using Anaconda, Graphviz is installed automatically via its conda-forge package. For other methods of installation, see the Graphviz download page for downloads and instructions for your system.

1.1.2 Install `ixmp`

Using Anaconda

After installing GAMS, we recommend that new users install Anaconda, and then use it to install *ixmp*. Advanced users may choose to install *ixmp* from source code (next section).

4. Install Python via either Miniconda or Anaconda.[1] We recommend the latest version; currently Python 3.10.[2]
5. Open a command prompt. We recommend Windows users use the "Anaconda Prompt" to avoid issues with permissions and environment variables when installing and using *ixmp*. This program is available in the Windows Start menu after installing Anaconda.
6. Configure conda to install *ixmp* from the conda-forge channel[3]:

```
$ conda config --prepend channels conda-forge
```

7. Create a new conda enviroment. This step is **required** if using Anaconda, but *optional* if using Miniconda. This example uses the name `ixmp_env`, but you can use any name of your choice:

```
$ conda create --name ixmp_env
$ conda activate ixmp_env
```

6. Install the `ixmp` package into the current environment (either `base`, or another name from step 7, e.g. `ixmp_env`):

```
$ conda install -c conda-forge ixmp
```

Note: When using Anaconda (not Miniconda), steps (5) through (8) can also be performed using the graphical Anaconda Navigator. See the Anaconda Navigator documentation for how to perform the various steps.

From source

4. (Optional) If you intend to contribute changes to *ixmp*, first register a Github account, and fork the ixmp repository. This will create a new repository `<user>/ixmp`. (Please also see Contributing to development.)
5. Clone either the main repository, or your fork; using the Github Desktop client, or the command line:

```
$ git clone git@github.com:iiasa/ixmp.git

# or:
$ git clone git@github.com:USER/ixmp.git
```

6. Open a command prompt in the `ixmp/` directory that is created, and type:

```
$ pip install --editable .[docs,tests,tutorial]
```

   The `--editable` flag ensures that changes to the source code are picked up every time `import ixmp` is used in Python code. The `[docs,tests,tutorial]` extra dependencies ensure additional dependencies are installed.

7. (Optional) Run the built-in test suite to check that *ixmp* functions correctly on your system:

[1] See the conda glossary for the differences between Anaconda and Miniconda, and the definitions of the terms 'channel' and 'environment' here.

[2] On newer macOS systems with "Apple M1" processors: the Miniconda or Anaconda installers provided for M1 lead to errors in ixmp. Instead, we recommend to use the macOS installers for "x86_64" processors on these systems. See also #473.

[3] The '$' character at the start of these lines indicates that the command text should be entered in the terminal or prompt, depending on the operating system. Do not retype the '$' character itself.

```
$ pytest
```

1.1.3 Troubleshooting

Run `ixmp show-versions` on the command line to check that you have all dependencies installed, or when reporting issues.

For Anaconda users experiencing problems during installation of ixmp, check that the following paths are part of the `PATH` environment variable, and add them if missing:

```
C:\[YOUR ANACONDA LOCATION]\Anaconda3;
C:\[YOUR ANACONDA LOCATION]\Anaconda3\Scripts;
C:\[YOUR ANACONDA LOCATION]\Anaconda3\Library\bin;
```

1.1.4 Install development tools

Developers making changes to the *ixmp* source **may** need one or more of the following tools. Users developing models using existing functionality **should not** need these tools.

Git

Use one of:

- https://git-scm.com/downloads
- https://desktop.github.com
- https://www.gitkraken.com

Java Development Kit (JDK)

- Install the Java Development Kit (JDK) for Java SE version 8 from https://www.oracle.com/technetwork/java/javase/downloads/index.html

  Note: At this point, ixmp is not compatible with JAVA SE 9.

- Follow the JDK website instructions to set the `JAVA_HOME` environment variable; if `JAVA_HOME` does not exist, add it as a new system variable.
- Update your `PATH` environment variable to point to the JRE binaries and server installation (e.g., `C:\Program Files\Java\jdk[YOUR JDK VERSION]\jre\bin\`, `C:\Program Files\Java\jdk[YOUR JDK VERSION]\jre\bin\server`).

  Warning: Do not overwrite the existing `PATH` environment variable, but add to the list of existing paths.

1.2 Tutorials

The tutorials provided in the `tutorial` directory of the *ixmp* repository walk through the first steps of working with *ixmp*. You will learn how to create a *ixmp.Scenario* as a structured data collection for sets, parameters and the numerical solution to the associated optimization problem.

1.2.1 Dantzig's transportation problem

We use Dantzig's transport problem [1], which is also used as the standard GAMS tutorial [4]. This problem finds a least cost shipping schedule that meets demand requirements at markets and supply capacity constraints at multiple factories.

The tutorials are provided as Jupyter notebooks for both Python and R, and are identical as far as possible.

- Tutorial 1: in Python, or in R.

  This tutorial walks through the following steps:

  1. Launch an *ixmp.Platform* instance and initialize a new *ixmp.Scenario*.
  2. Define the sets and parameters in the scenario, and commit the data to the platform.
  3. Check out the scenario and initialize variables and equations (necessary for ixmp to import the solution).
  4. Solve the scenario (export to GAMS input gdx, execute, read solution from output gdx).
  5. Display the solution (variables and equation).

- Tutorial 2: in Python, or in R.

  This tutorial creates an alternate or 'counterfactual' scenario of the transport problem; solves it; and compares the results to the original or reference scenario.

1.2.2 Adapting GAMS models for `ixmp`

The common example optimization from Dantzig [1] is available in a GAMS implementation from the GAMS website. The file `tutorial/transport/transport_ixmp.gms` illustrates how an existing GAMS model can be adapted to work with the *ix modeling platform*. The same, simple procedure can be applied to any GAMS code.

The steps are:

1. Modify the definitions of GAMS sets (`i` and `j`) and parameters (`a`, `b`, `d`, and `f`) to **remove explicit values**.
2. Add lines to **read the model input data passed by ixmp**.

   The following lines are added *before* the code that defines and solves the model:

   ```
   * These two lines let the model code be run outside of ixmp, if needed
   $if not set in  $setglobal in  'ix_transport_data.gdx'
   $if not set out $setglobal out 'ix_transport_results.gdx'

   $gdxin '%in%'
   $load i, j, a, b, d, f
   $gdxin
   ```

3. Add a line to **write the model output data**.

   The following line is added *after* the model's `solve ...;` statement:

```
execute_unload '%out%';
```

ixmp's *GAMSModel* class uses command-line options to pass the values of the variables `in` and `out`. This causes the model to read its input data from a GDX-format file created by *ixmp*, and write its output data to a GDX file specified by *ixmp*. *ixmp* then automatically retrieves the model solution and other information from the output file, updating the *Scenario* and storage *Backend*.

1.3 Command-line interface

ixmp has a **command-line** interface:

```
$ ixmp --help
Usage: ixmp [OPTIONS] COMMAND [ARGS]...

Options:
  --url ixmp://PLATFORM/MODEL/SCENARIO[#VERSION]
                                  Scenario URL.
  --platform TEXT                 Configured platform name.
  --dbprops FILE                  Database properties file.
  --model TEXT                    Model name.
  --scenario TEXT                 Scenario name.
  --version VERSION               Scenario version.
  --help                          Show this message and exit.

Commands:
  config         Get and set configuration keys.
  export         Export scenario data to PATH.
  import         Import time series or scenario data.
  list           List scenarios on the --platform.
  platform       Configure platforms and storage backends.
  report         Run reporting for KEY.
  show-versions  Print versions of ixmp and its dependencies.
  solve          Solve a Scenario and store results on the Platform.
```

The various commands allow to manipulate *Configuration*, show debug and system information, invoke particular models and the *Reporting / postprocessing* features. The CLI is used as the basis for extended features provided by `message_ix`.

1.3.1 CLI internals

class ixmp.cli.**VersionType**

A Click parameter type that accepts `int` or 'all'.

convert(*value*, *param*, *ctx*)

Fail if *value* is not `int` or 'all'.

name: str = 'version'

the descriptive name of this type

- *Installation*
- *Tutorials*

- *Command-line interface*

CHAPTER

TWO

SCIENTIFIC PROGRAMMING API

2.1 Python (`ixmp` package)

The *ix modeling platform* application programming interface (API) is organized around three classes:

Platform([name, backend])	Instance of the modeling platform.
TimeSeries(mp, model, scenario[, version, ...])	Collection of data in time series format.
Scenario(mp, model, scenario[, version, ...])	Collection of model-related data.

2.1.1 Platform

class ixmp.**Platform**(*name: str | None = None*, *backend: str | None = None*, ***backend_args*)

Instance of the modeling platform.

A Platform connects two key components:

1. A **back end** for storing data such as model inputs and outputs.
2. One or more **model(s)**; codes in Python or other languages or frameworks that run, via *Scenario.solve()*, on the data stored in the Platform.

The Platform parameters control these components. *TimeSeries* and *Scenario* objects tied to a single Platform; to move data between platforms, see *Scenario.clone()*.

Parameters

- **name** (str) – Name of a specific *configured* backend.
- **backend** ('jdbc') – Storage backend type. 'jdbc' corresponds to the built-in *JDBCBackend*; see *BACKENDS*.
- **backend_args** – Keyword arguments to specific to the *backend*. See *JDBCBackend*.

Platforms have the following methods:

add_region(region, hierarchy[, parent])	Define a region including a hierarchy level and a 'parent' region.
add_region_synonym(region, mapped_to)	Define a synonym for a *region*.
add_unit(unit[, comment])	Define a unit.
check_access(user, models[, access])	Check access to specific models.
regions()	Return all regions defined time series data, including synonyms.
scenario_list([default, model, scen])	Return information about TimeSeries and Scenarios on the Platform.
set_log_level(level)	Set log level for the Platform and its storage Backend.
units()	Return all units defined on the Platform.

The following backend methods are available via Platform too:

backend.base.Backend.add_model_name(name)	Add (register) new model name.
backend.base.Backend.add_scenario_name(name)	Add (register) new scenario name.
backend.base.Backend.close_db()	OPTIONAL: Close database connection(s).
backend.base.Backend.get_doc(domain[, name])	Read documentation from database
backend.base.Backend.get_meta(model, ...)	Retrieve all metadata attached to a specific target.
backend.base.Backend.get_model_names()	List existing model names.
backend.base.Backend.get_scenario_names()	List existing scenario names.
backend.base.Backend.open_db()	OPTIONAL: (Re-)open database connection(s).
backend.base.Backend.remove_meta(names, ...)	Remove metadata attached to a target.
backend.base.Backend.set_doc(domain, docs)	Save documentation to database
backend.base.Backend.set_meta(meta, model, ...)	Set metadata on a target.

These methods can be called like normal Platform methods, e.g.:

```
$ platform_instance.close_db()
```

add_region(*region: str*, *hierarchy: str*, *parent: str = 'World'*) → None

Define a region including a hierarchy level and a 'parent' region.

> **Tip:** On a *Platform* backed by a shared database, a region may already exist with a different spelling. Use *regions()* first to check, and consider calling *add_region_synonym()* instead.

Parameters

- **region** (str) – Name of the region.
- **hierarchy** (str) – Hierarchy level of the region (e.g., country, R11, basin)
- **parent** (str, *optional*) – Assign a 'parent' region.

add_region_synonym(*region: str*, *mapped_to: str*) → None

Define a synonym for a *region*.

When adding timeseries data using the synonym in the region column, it will be converted to *mapped_to*.

Parameters

- **region** (str) – Name of the region synonym.
- **mapped_to** (str) – Name of the region to which the synonym should be mapped.

add_timeslice(*name: str*, *category: str*, *duration: float*) → None

Define a subannual timeslice including a category and duration.

See *timeslices()* for a detailed description of timeslices.

Parameters

- **name** (str) – Unique name of the timeslice.
- **category** (str) – Timeslice category (e.g. 'common', 'month', etc).
- **duration** (float) – Duration of timeslice as fraction of year.

add_unit(*unit: str*, *comment: str = 'None'*) → None

Define a unit.

Parameters

- **unit** (str) – Name of the unit.
- **comment** (str, *optional*) – Annotation describing the unit or why it was added. The current database user and timestamp are appended automatically.

check_access(*user: str*, *models: str | Sequence[str]*, *access: str = 'view'*) → bool | Dict[str, bool]

Check access to specific models.

Parameters

- **user** (str) – Registered user name
- **models** (str or list of str) – Model(s) name
- **access** (str, *optional*) – Access type - view or edit

Return type

bool or dict of bool

export_timeseries_data(*path: PathLike*, *default: bool = True*, *model: str | None = None*, *scenario: str | None = None*, *variable=None*, *unit=None*, *region=None*, *export_all_runs: bool = False*) → None

Export time series data to CSV file across multiple *TimeSeries*.

Refer *TimeSeries.add_timeseries()* about adding time series data.

Parameters

- **path** (os.PathLike) – File name to export data to; must have the suffix '.csv'.

  Result file will contain the following columns:

  – model

  – scenario

  – version

  - variable
  - unit
  - region
  - meta
  - subannual
  - year
  - value
- **default** (bool, *optional*) – True to include only TimeSeries versions marked as default.
- **model** (str, *optional*) – Only return data for this model name.
- **scenario** (str, *optional*) – Only return data for this scenario name.
- **variable** (list of str, *optional*) – Only return data for variable name(s) in this list.
- **unit** (list of str, *optional*) – Only return data for unit name(s) in this list.
- **region** (list of str, *optional*) – Only return data for region(s) in this list.
- **export_all_runs** (bool, *optional*) – Export all existing model+scenario run combinations.

get_log_level() → str

Return log level of the storage *Backend*, if any.

Returns
Name of a Python logging level.

Return type
str

regions() → DataFrame

Return all regions defined time series data, including synonyms.

Return type
pandas.DataFrame

scenario_list(*default: bool = True*, *model: str | None = None*, *scen: str | None = None*) → DataFrame

Return information about TimeSeries and Scenarios on the Platform.

Parameters

- **default** (bool, *optional*) – Return *only* the default version of each TimeSeries/Scenario (see *TimeSeries.set_as_default()*). Any (*model*, *scenario*) without a default version is omitted. If False, return all versions.
- **model** (str, *optional*) – A model name. If given, only return information for *model*.
- **scen** (str, *optional*) – A scenario name. If given, only return information for *scen*.

Returns

Scenario information, with the columns:

- model, scenario, version, and scheme—Scenario identifiers; see *TimeSeries* and *Scenario*.
- is_default—True if the version is the default version for the (model, scenario).
- is_locked—True if the Scenario has been locked for use.

- `cre_user`, `cre_date`—database user that created the Scenario, and creation time.
- `upd_user`, `upd_date`—user and time for last modification of the Scenario.
- `lock_user`, `lock_date`—user that locked the Scenario and lock time.
- `annotation`: description of the Scenario or changelog.

Return type
`pandas.DataFrame`

set_log_level(*level: str | int*) → None

Set log level for the Platform and its storage *Backend*.

Parameters
level (str) – Name of a Python logging level.

timeslices() → DataFrame

Return all subannual time slices defined in this Platform instance.

See the *Data model* documentation for further details.

The category and duration do not have any functional relevance within the ixmp framework, but they may be useful for pre- or post-processing. For example, they can be used to filter all timeslices of a certain category (e.g., all months) from the `pandas.DataFrame` returned by this function or to aggregate subannual data to full-year results.

Returns
Data frame with columns 'timeslice', 'category', and 'duration'.

Return type
`pandas.DataFrame`

See also:

add_timeslice

units() → List[str]

Return all units defined on the Platform.

Return type
`list` of `str`

2.1.2 TimeSeries

class ixmp.**TimeSeries**(*mp: Platform*, *model: str*, *scenario: str*, *version: str | int | None = None*, *annotation: str | None = None*, ***kwargs*)

Collection of data in time series format.

TimeSeries is the parent/super-class of *Scenario*.

Parameters

- **mp** (*Platform*) – ixmp instance in which to store data.
- **model** (str) – Model name.
- **scenario** (str) – Scenario name.
- **version** (int or str, *optional*) – If omitted and a default version of the (*model*, *scenario*) has been designated (see *set_as_default()*), load that version. If int, load a specific version. If `'new'`, create a new TimeSeries.

- **annotation** (str, *optional*) – A short annotation/comment used when `version='new'`.

A TimeSeries is uniquely identified on its *Platform* by its *model*, *scenario*, and *version* attributes. For more details, see the *data model documentation*.

A new *version* is created by:

- Instantiating a new TimeSeries with the same *model* and *scenario* as an existing TimeSeries.
- Calling *Scenario.clone()*.

TimeSeries objects have the following methods and attributes:

add_geodata(df)	Add geodata.
add_timeseries(df[, meta, year_lim])	Add time series data.
check_out([timeseries_only])	Check out the TimeSeries.
commit(comment)	Commit all changed data to the database.
discard_changes()	Discard all changes and reload from the database.
get_geodata()	Fetch geodata and return it as dataframe.
get_meta([name])	Get *Metadata* for this object.
is_default()	Return `True` if the *version* is the default version.
last_update()	Get the timestamp of the last update/edit of this TimeSeries.
preload_timeseries()	Preload timeseries data to in-memory cache.
read_file(path[, firstyear, lastyear])	Read time series data from a CSV or Microsoft Excel file.
remove_geodata(df)	Remove geodata from the TimeSeries instance.
remove_timeseries(df)	Remove time series data.
run_id()	Get the run id of this TimeSeries.
set_as_default()	Set the current *version* as the default.
set_meta(name_or_dict[, value])	Set *Metadata* for this object.
timeseries([region, variable, unit, year, ...])	Retrieve time series data.
transact([message, condition, discard_on_error])	Context manager to wrap code in a 'transaction'.
url	URL fragment for the TimeSeries.

add_geodata(*df: DataFrame*) → None

Add geodata.

Parameters

df (pandas.DataFrame) – Data to add. *df* must have the following columns:

- *region*
- *variable*
- *subannual*
- *unit*
- *year*
- *value*
- *meta*

add_timeseries(*df: DataFrame*, *meta: bool = False*, *year_lim: Tuple[int | None, int | None] = (None, None)*) → None

Add time series data.

Parameters

- **df** (pandas.DataFrame) – Data to add. *df* must have the following columns:
  - *region* or *node*
  - *variable*
  - *unit*

  Additional column names may be either of:
  - *year* and *value*—long, or 'tabular', format.
  - one or more specific years—wide, or 'IAMC' format.

  To support subannual temporal resolution of timeseries data, a column *subannual* is optional in *df*. The entries in this column must have been defined in the Platform instance using *add_timeslice()* beforehand. If no column *subannual* is included in *df*, the data is assumed to contain yearly values. See *timeslices()* for a detailed description of the feature.
- **meta** (bool, *optional*) – If True, store *df* as metadata. Metadata is treated specially when *Scenario.clone()* is called for Scenarios created with scheme='MESSAGE'.
- **year_lim** (tuple, *optional*) – Respectively, earliest and latest years to add from *df*; data for other years is ignored.

check_out(*timeseries_only: bool = False*) → None

Check out the TimeSeries.

Data in the TimeSeries can only be modified when it is in a checked-out state.

See also:

util.maybe_check_out

commit(*comment: str*) → None

Commit all changed data to the database.

If the TimeSeries was newly created (with version='new'), *version* is updated with a new version number assigned by the backend. Otherwise, *commit()* does not change the *version*.

Parameters
comment (str) – Description of the changes being committed.

See also:

util.maybe_commit

delete_meta(**args*, ***kwargs*) → None

Remove *Metadata* for this object.

Deprecated since version 3.1: Use *remove_meta()*.

Parameters
name (str or list of str) – Either single metadata name/identifier, or list of names.

discard_changes() → None

Discard all changes and reload from the database.

classmethod from_url(*url: str*, *errors: Literal['warn', 'raise'] = 'warn'*) → Tuple[*TimeSeries* | None, *Platform*]

Instantiate a TimeSeries (or Scenario) given an ixmp:// URL.

The following are equivalent:

```
from ixmp import Platform, TimeSeries
mp = Platform(name='example')
scen = TimeSeries(mp 'model', 'scenario', version=42)
```

and:

```
from ixmp import TimeSeries
scen, mp = TimeSeries.from_url('ixmp://example/model/scenario#42')
```

Parameters

- **url** (str) – See *parse_url*.
- **errors** ('warn' or 'raise') – If 'warn', a failure to load the TimeSeries is logged as a warning, and the platform is still returned. If 'raise', the exception is raised.

Returns

with 2 elements:

- The *TimeSeries* referenced by the *url*.
- The *Platform* referenced by the *url*, on which the first element is stored.

Return type
tuple

get_geodata() → DataFrame

Fetch geodata and return it as dataframe.

Returns
Specified data.

Return type
pandas.DataFrame

get_meta(*name: str | None = None*)

Get *Metadata* for this object.

Metadata with the given *name*, attached to this (*model* name, *scenario* name, *version*), is retrieved.

Parameters
name (str, *optional*) – Metadata name/identifier.

is_default() → bool

Return True if the *version* is the default version.

last_update() → str

Get the timestamp of the last update/edit of this TimeSeries.

model: str

Name of the model associated with the TimeSeries.

preload_timeseries() → None

Preload timeseries data to in-memory cache. Useful for bulk updates.

read_file(*path: PathLike*, *firstyear: int | None = None*, *lastyear: int | None = None*) → None

Read time series data from a CSV or Microsoft Excel file.

Parameters

- **path** (os.PathLike) – File to read. Must have suffix '.csv' or '.xlsx'.
- **firstyear** (int, *optional*) – Only read data from years equal to or later than this year.
- **lastyear** (int, *optional*) – Only read data from years equal to or earlier than this year.

See also:

Scenario.read_excel

remove_geodata(*df: DataFrame*) → None

Remove geodata from the TimeSeries instance.

Parameters

df (pandas.DataFrame) – Data to remove. *df* must have the following columns:

- *region*
- *variable*
- *unit*
- *subannual*
- *year*

remove_meta(*name: str | Sequence[str]*) → None

Remove *Metadata* for this object.

Parameters

name (str or list of str) – Either single metadata name/identifier, or list of names.

remove_timeseries(*df: DataFrame*) → None

Remove time series data.

Parameters

df (pandas.DataFrame) – Data to remove. *df* must have the following columns:

- *region* or *node*
- *variable*
- *unit*
- *year*

run_id() → int

Get the run id of this TimeSeries.

scenario: str

Name of the scenario associated with the TimeSeries.

set_as_default() → None

Set the current *version* as the default.

set_meta(*name_or_dict: str | Dict[str, Any]*, *value=None*) → None

Set *Metadata* for this object.

Parameters

- **name_or_dict** (str or dict) – If dict, a mapping of names/identifiers to values. Otherwise, use the metadata identifier.
- **value** (str or float or int or bool, *optional*) – Metadata value.

timeseries(*region: str | Sequence[str] | None = None*, *variable: str | Sequence[str] | None = None*, *unit: str | Sequence[str] | None = None*, *year: int | Sequence[int] | None = None*, *iamc: bool = False*, *subannual: bool | str = 'auto'*) → DataFrame

Retrieve time series data.

Parameters

- **iamc** (`bool`, *optional*) – Return data in wide/'IAMC' format. If `False`, return data in long format; see *add_timeseries()*.
- **region** (`str` or `list` of `str`, *optional*) – Regions to include in returned data.
- **variable** (`str` or `list` of `str`, *optional*) – Variables to include in returned data.
- **unit** (`str` or `list` of `str`, *optional*) – Units to include in returned data.
- **year** (`int` or `list` of `int`, *optional*) – Years to include in returned data.
- **subannual** (`bool` or `'auto'`, *optional*) – Whether to include column for sub-annual specification (if `bool`); if 'auto', include column if sub-annual data (other than 'Year') exists in returned data frame.

Raises

ValueError – If *subannual* is `False` but Scenario has (filtered) sub-annual data.

Returns

Specified data.

Return type

`pandas.DataFrame`

transact(*message: str = ''*, *condition: bool = True*, *discard_on_error: bool = False*)

Context manager to wrap code in a 'transaction'.

Parameters

- **message** (`str`) – Commit message to use, if any commit is performed.
- **condition** (`bool`) – If `True` (the default):
  - Before entering the code block, the TimeSeries (or *Scenario*) is checked out.
  - On exiting the code block normally (without an exception), changes are committed with *message*.

  If `False`, nothing occurs on entry or exit.
- **discard_on_error** (`bool`) – If `True` (default `False`), then the anti-locking behaviour of *discard_on_error()* also applies to any exception raised in the block.

Example

```
>>> # `ts` is currently checked in/locked
>>> with ts.transact(message="replace 'foo' with 'bar' in set x"):
>>>     # `ts` is checked out and may be modified
>>>     ts.remove_set("x", "foo")
>>>     ts.add_set("x", "bar")
>>> # Changes to `ts` have been committed
```

property url: *str*

URL fragment for the TimeSeries.

This has the format `{model name}/{scenario name}#{version}`, with the same values passed when creating the TimeSeries instance.

Examples

To form a complete URL (e.g. to use with *from_url()*), use a configured *Platform* name:

```
>>> platform_name = "my-ixmp-platform"
>>> mp = Platform(platform_name)
>>> ts = TimeSeries(mp, "foo", "bar", 34)
>>> ts.url
"foo/bar#34"
>>> f"ixmp://{platform_name}/{ts.url}"
"ixmp://platform_name/foo/bar#34"
```

Note: Use caution: because Platform configuration is system-specific, other systems must have the same configuration for *platform_name* in order for the URL to refer to the same TimeSeries/Scenario.

version = None

Version of the TimeSeries. Immutable for a specific instance.

2.1.3 Scenario

class ixmp.Scenario(*mp:* Platform, *model: str*, *scenario: str*, *version: str | int | None = None*, *scheme: str | None = None*, *annotation: str | None = None*, ***model_init_args*)

Bases: *TimeSeries*

Collection of model-related data.

See *TimeSeries* for the meaning of parameters *mp*, *model*, *scenario*, *version*, and *annotation*.

Parameters

- **scheme** (*str*, *optional*) – Use an explicit scheme to initialize the new scenario. The *initialize()* method of the corresponding *Model* class in *MODELS* is used to initialize items in the Scenario.
- **cache** – Deprecated since version 3.0: The *cache* keyword argument to *Scenario* has no effect and raises a warning. Use *cache* as one of the *backend_args* to *Platform* to disable/enable caching for storage backends that support it. Use *load_scenario_data()* to load all data in the Scenario into an in-memory cache.

A Scenario is a *TimeSeries* that also contains model data, including model solution data. See the *data model documentation*.

The Scenario class provides methods to manipulate *model data items*. In addition to generic methods (*init_item()*, *items()*, *list_items()*, *has_item()*), there are methods for each of the four item types:

- Set: *init_set()*, *add_set()*, *set()*, *remove_set()*, *has_set()*
- Parameter:
  - 1-dimensional: *init_par()*, *add_par()*, *par()*, *remove_par()*, *par_list()*, and *has_par()*.

  - 0-dimensional: *init_scalar()*, *change_scalar()*, and *scalar()*. These are thin wrappers around the corresponding *_par methods, which can also be used to manipulate 0-dimensional parameters.

- Variable: *init_var()*, *var()*, *var_list()*, and *has_var()*.
- Equation: *init_equ()*, *equ()*, *equ_list()*, and *has_equ()*.

add_par(name[, key_or_data, value, unit, ...])	Set the values of a parameter.
add_set(name, key[, comment])	Add elements to an existing set.
change_scalar(name, val, unit[, comment])	Set the value and unit of a scalar.
clone([model, scenario, annotation, ...])	Clone the current scenario and return the clone.
equ(name[, filters])	Return a dataframe of (filtered) elements for a specific equation.
has_item(name[, item_type])	Check whether the Scenario has an item *name* of *item_type*.
has_solution()	Return `True` if the Scenario contains model solution data.
idx_names(name)	Return the list of index names for an item (set, par, var, equ).
idx_sets(name)	Return the list of index sets for an item (set, par, var, equ).
init_item(item_type, name[, idx_sets, idx_names])	Initialize a new item *name* of type *item_type*.
init_scalar(name, val, unit[, comment])	Initialize a new scalar and set its value.
items([type, filters, indexed_by, par_data])	Iterate over model data items.
list_items(item_type[, indexed_by])	List all defined items of type *item_type*.
load_scenario_data()	Load all Scenario data into memory.
par(name[, filters])	Return parameter data.
read_excel(path[, add_units, init_items, ...])	Read a Microsoft Excel file into the Scenario.
remove_par(name[, key])	Remove parameter values or an entire parameter.
remove_set(name[, key])	Delete set elements or an entire set.
remove_solution([first_model_year])	Remove the solution from the scenario.
scalar(name)	Return the value and unit of a scalar.
set(name[, filters])	Return the (filtered) elements of a set.
solve([model, callback, cb_kwargs])	Solve the model and store output.
to_excel(path[, items, filters, max_row])	Write Scenario to a Microsoft Excel file.
var(name[, filters])	Return a dataframe of (filtered) elements for a specific variable.

add_par(*name: str*, *key_or_data: str | Sequence[str] | Dict | DataFrame | None = None*, *value=None*, *unit: str | None = None*, *comment: str | None = None*) → None

Set the values of a parameter.

Parameters

- **name** (`str`) – Name of the parameter.
- **key_or_data** (`str` or `collections.abc.Iterable` of `str` or `range` or `dict` or `pandas.DataFrame`) – Element(s) to be added.
- **value** (`float` or `collections.abc.Iterable` of `float`, *optional*) – Values.
- **unit** (`str` or `collections.abc.Iterable` of `str`, *optional*) – Unit symbols.
- **comment** (`str` or `collections.abc.Iterable` of `str`, *optional*) – Comment(s) for the added values.

add_set(*name: str*, *key: str | Sequence[str] | Dict | DataFrame*, *comment: str | None = None*) → None

Add elements to an existing set.

Parameters

- **name** (str) – Name of the set.
- **key** (str or collections.abc.Iterable of str or dict or pandas.DataFrame) – Element(s) to be added. If *name* exists, the elements are appended to existing elements.
- **comment** (str or collections.abc.Iterable of str, *optional*) – Comment describing the element(s). If given, there must be the same number of comments as elements.

Raises

- **KeyError** – If the set *name* does not exist. *init_set()* must be called before *add_set()*.
- **ValueError** – For invalid forms or combinations of *key* and *comment*.

change_scalar(*name: str*, *val: Real*, *unit: str*, *comment: str | None = None*) → None

Set the value and unit of a scalar.

Parameters

- **name** (str) – Name of the scalar.
- **val** (float or int) – New value of the scalar.
- **unit** (str) – New unit of the scalar.
- **comment** (str, *optional*) – Description of the change.

check_out(*timeseries_only: bool = False*) → None

Check out the Scenario.

Raises

ValueError – If *has_solution()* is True.

See also:

TimeSeries.check_out, *util.maybe_check_out*

clone(*model: str | None = None*, *scenario: str | None = None*, *annotation: str | None = None*, *keep_solution: bool = True*, *shift_first_model_year: int | None = None*, *platform: Platform | None = None*) → *Scenario*

Clone the current scenario and return the clone.

If the (*model*, *scenario*) given already exist on the *Platform*, the *version* for the cloned Scenario follows the last existing version. Otherwise, the *version* for the cloned Scenario is 1.

Note: *clone()* does not set or alter default versions. This means that a clone to new (*model*, *scenario*) names has no default version, and will not be returned by *Platform.scenario_list()* unless *default=False* is given.

Parameters

- **model** (str, *optional*) – New model name. If not given, use the existing model name.
- **scenario** (str, *optional*) – New scenario name. If not given, use the existing scenario name.

- **annotation** (str, *optional*) – Explanatory comment for the clone commit message to the database.
- **keep_solution** (bool, *optional*) – If True, include all timeseries data and the solution (vars and equs) from the source scenario in the clone. If False, only include timeseries data marked *meta=True* (see *add_timeseries()*).
- **shift_first_model_year** (int, *optional*) – If given, all timeseries data in the Scenario is omitted from the clone for years from *first_model_year* onwards. Timeseries data with the *meta* flag (see *add_timeseries()*) are cloned for all years.
- **platform** (*Platform*, *optional*) – Platform to clone to (default: current platform)

equ(*name: str*, *filters=None*, ***kwargs*) → DataFrame

Return a dataframe of (filtered) elements for a specific equation.

Parameters

- **name** (str) – name of the equation
- **filters** (dict) – index names mapped list of index set elements

equ_list(*indexed_by: str | None = None*) → List[str]

List all defined equations. See *list_items()*.

has_equ(*name: str*, *, *item_type=ItemType.EQU*) → bool

Check whether the scenario has a equation *name*. See *has_item()*.

has_item(*name: str*, *item_type=ItemType.MODEL*) → bool

Check whether the Scenario has an item *name* of *item_type*.

In general, user code **should** call one of *has_equ()*, *has_par()*, *has_set()*, or *has_var()* instead of calling this method directly.

Returns

- True – if the Scenario contains an item of *item_type* with name *name*.
- False – otherwise

See also:

items

has_par(*name: str*, *, *item_type=ItemType.PAR*) → bool

Check whether the scenario has a parameter *name*. See *has_item()*.

has_set(*name: str*, *, *item_type=ItemType.SET*) → bool

Check whether the scenario has a set *name*. See *has_item()*.

has_solution() → bool

Return True if the Scenario contains model solution data.

has_var(*name: str*, *, *item_type=ItemType.VAR*) → bool

Check whether the scenario has a variable *name*. See *has_item()*.

idx_names(*name: str*) → List[str]

Return the list of index names for an item (set, par, var, equ).

Parameters

name (str) – name of the item

idx_sets(*name: str*) → List[str]

Return the list of index sets for an item (set, par, var, equ).

Parameters
name (str) – name of the item

init_equ(*name: str*, *idx_sets: Sequence[str] | None = None*, *idx_names: Sequence[str] | None = None*)

Initialize a new equation. See *init_item()*.

init_item(*item_type: ItemType*, *name: str*, *idx_sets: Sequence[str] | None = None*, *idx_names: Sequence[str] | None = None*)

Initialize a new item *name* of type *item_type*.

In general, user code **should** call one of *init_set()*, *init_par()*, *init_var()*, or *init_equ()* instead of calling this method directly.

Parameters

- **item_type** (*ItemType*) – The type of the item.
- **name** (str) – Name of the item.
- **idx_sets** (collections.abc.Sequence of str or str, *optional*) – Name(s) of index sets for a 1+-dimensional item. If none are given, the item is scalar (zero dimensional).
- **idx_names** (collections.abc.Sequence of str or str, *optional*) – Names of the dimensions indexed by *idx_sets*. If given, they must be the same length as *idx_sets*.

Raises

- **ValueError** –
  - if *idx_names* are given but do not match the length of *idx_sets*. - if an item with the same *name*, of any *item_type*, already exists.
- **RuntimeError** – if the Scenario is not checked out (see *check_out()*).

init_par(*name: str*, *idx_sets: Sequence[str] | None = None*, *idx_names: Sequence[str] | None = None*)

Initialize a new parameter. See *init_item()*.

init_scalar(*name: str*, *val: Real*, *unit: str*, *comment=None*) → None

Initialize a new scalar and set its value.

Parameters

- **name** (str) – Name of the scalar
- **val** (float or int) – Initial value of the scalar.
- **unit** (str) – Unit of the scalar.
- **comment** (str, *optional*) – Description of the scalar.

init_set(*name: str*, *idx_sets: Sequence[str] | None = None*, *idx_names: Sequence[str] | None = None*)

Initialize a new set. See *init_item()*.

init_var(*name: str*, *idx_sets: Sequence[str] | None = None*, *idx_names: Sequence[str] | None = None*)

Initialize a new variable. See *init_item()*.

items(*type: ItemType = ItemType.PAR*, *filters: Dict[str, Sequence[str]] | None = None*, ***, *indexed_by: str | None = None*, *par_data: bool | None = None*) → Iterable[str]

Iterate over model data items.

Parameters

- **type** (*ItemType*, *optional*) – Types of items to iterate, for instance *ItemType.PAR* for parameters.
- **filters** (dict, *optional*) – Filters for values along dimensions; same as the *filters* argument to *par()*. Only value for *ItemType.PAR*.
- **indexed_by** (str, *optional*) – If given, only iterate over items where one of the item dimensions is *indexed_by* the set of this name.
- **par_data** (bool, *optional*) – If True (the default) and *type* is *ItemType.PAR*, also iterate over data for each parameter.

Yields

- str – if *type* is not *ItemType.PAR*, or *par_data* is False: names of items.
- tuple – if *type* is *ItemType.PAR* and *par_data* is True: each tuple is (item name, item data).

list_items(*item_type:* ItemType, *indexed_by: str | None = None*) → List[str]

List all defined items of type *item_type*.

See also:

items

load_scenario_data() → None

Load all Scenario data into memory.

Raises

ValueError – If the Scenario was instantiated with cache=False.

par(*name: str*, *filters: Dict[str, Sequence[str]] | None = None*, ***kwargs*) → DataFrame

Return parameter data.

If *filters* is provided, only a subset of data, matching the filters, is returned.

Parameters

- **name** (str) – Name of the parameter
- **filters** (dict, *optional*) – Keys are index names. Values are lists of index set elements. Elements not appearing in the respective index set(s) are silently ignored.

par_list(*indexed_by: str | None = None*) → List[str]

List all defined parameters. See *list_items()*.

read_excel(*path: PathLike*, *add_units: bool = False*, *init_items: bool = False*, *commit_steps: bool = False*) → None

Read a Microsoft Excel file into the Scenario.

Parameters

- **path** (os.PathLike) – File to read. Must have suffix '.xlsx'.
- **add_units** (bool, *optional*) – Add missing units, if any, to the Platform instance.
- **init_items** (bool, *optional*) – Initialize sets and parameters that do not already exist in the Scenario.
- **commit_steps** (bool, *optional*) – Commit changes after every data addition.

See also:

Scenario/model data, *TimeSeries.read_file*, *to_excel*

remove_par(*name: str*, *key=None*) → None

Remove parameter values or an entire parameter.

Parameters

- **name** (str) – Name of the parameter.
- **key** (pandas.DataFrame or list or str, *optional*) – Elements to be removed. If a pandas.DataFrame, must contain the same columns (indices/dimensions) as the parameter. If a list, a single key for a single data point; the individual elements must correspond to the indices/dimensions of the parameter.

remove_set(*name: str*, *key: str | Sequence[str] | Dict | DataFrame | None = None*) → None

Delete set elements or an entire set.

Parameters

- **name** (str) – Name of the set to remove (if *key* is None) or from which to remove elements.
- **key** (pandas.DataFrame or list of str, *optional*) – Elements to be removed from set *name*.

remove_solution(*first_model_year: int | None = None*) → None

Remove the solution from the scenario.

This function removes the solution (variables and equations) and timeseries data marked as *meta=False* from the scenario (see *add_timeseries()*).

Parameters

first_model_year (int, *optional*) – If given, timeseries data marked as *meta=False* is removed only for years from *first_model_year* onwards.

Raises

ValueError – If Scenario has no solution or if *first_model_year* is not *int*.

scalar(*name: str*) → Dict[str, Real | str]

Return the value and unit of a scalar.

Parameters

name (str) – Name of the scalar.

Returns

with the keys “value” and “unit”.

Return type

dict

scheme = None

Scheme of the Scenario.

set(*name: str*, *filters: Dict[str, Sequence[str]] | None = None*, ***kwargs*) → List[str] | DataFrame

Return the (filtered) elements of a set.

Parameters

- **name** (str) – Name of the set.
- **filters** (dict) – Mapping of *dimension_name* → *elements*, where *dimension_name* is one of the *idx_names* given when the set was initialized (see *init_set()*), and *elements* is an iterable of labels to include in the return value.

Return type

pandas.DataFrame

set_list(*indexed_by: str | None = None*) → List[str]

List all defined sets. See `list_items()`.

solve(*model: str | None = None*, *callback: Callable | None = None*, *cb_kwargs: Dict[str, Any] = {}*, ***model_options*) → None

Solve the model and store output.

ixmp 'solves' a model by invoking the run() method of a `Model` subclass—for instance, `GAMSModel.run()`. Depending on the underlying model code, different steps are taken; see each model class for details. In general:

1. Data from the Scenario are written to a **model input file**.
2. Code or an external program is invoked to perform calculations or optimizations, **solving the model**.
3. Data representing the model outputs or solution are read from a **model output file** and stored in the Scenario.

If the optional argument *callback* is given, additional steps are performed:

4. Execute the *callback* with the Scenario as an argument. The Scenario has an *iteration* attribute that stores the number of times the underlying model has been solved (#2).
5. If the *callback* returns `False` or similar, iterate by repeating from step #1. Otherwise, exit.

Parameters

- **model** (`str`) – model (e.g., MESSAGE) or GAMS file name (excluding '.gms')
- **callback** (callable, *optional*) – Method to execute arbitrary non-model code. Must accept a single argument: the Scenario. Must return a non-`False` value to indicate convergence.
- **cb_kwargs** (`dict`, *optional*) – Keyword arguments to pass to *callback*.
- **model_options** – Keyword arguments specific to the *model*. See `GAMSModel`.

Warns

UserWarning – If *callback* is given and returns `None`. This may indicate that the user has forgotten a `return` statement, in which case the iteration will continue indefinitely.

Raises

`ValueError` – If the Scenario has already been solved.

to_excel(*path: PathLike*, *items: ItemType = ItemType.SET | PAR*, *filters: Dict[str, Sequence[str]] | Scenario] | None = None*, *max_row: int | None = None*) → None

Write Scenario to a Microsoft Excel file.

Parameters

- **path** (`os.PathLike`) – File to write. Must have suffix `.xlsx`.
- **items** (`ItemType`, *optional*) – Types of items to write. Either `SET` | `PAR` (i.e. only sets and parameters), or `MODEL` (also variables and equations, i.e. model solution data).
- **filters** (`dict`, *optional*) – Filters for values along dimensions; same as the *filters* argument to `par()`.
- **max_row** (`int`, *optional*) – Maximum number of rows in each sheet. If the number of elements in an item exceeds this number or `EXCEL_MAX_ROWS`, then an item is written to multiple sheets named, e.g. 'foo', 'foo(2)', 'foo(3)', etc.

See also:

Scenario/model data, `read_excel`

var(*name: str*, *filters=None*, ***kwargs*)

Return a dataframe of (filtered) elements for a specific variable.

Parameters

- **name** (str) – name of the variable
- **filters** (dict) – index names mapped list of index set elements

var_list(*indexed_by: str | None = None*) → List[str]

List all defined variables. See *list_items()*.

2.1.4 Configuration

When imported, *ixmp* reads configuration from the first file named `config.json` found in one of the following directories:

1. The directory given by the environment variable `IXMP_DATA`, if defined,
2. `${XDG_DATA_HOME}/ixmp`, if the environment variable is defined, or
3. `$HOME/.local/share/ixmp`.

Tip: For most users, #2 or #3 is a sensible default; platform information for many local and remote databases can be stored in `config.json` and retrieved by name.

Advanced users wishing to use a project-specific `config.json` can set `IXMP_DATA` to the path for any directory containing a file with this name.

To manipulate the configuration file, use the `platform` command in the ixmp command-line interface:

```
# Add a platform named 'p1' backed by a local HSQL database
$ ixmp platform add p1 jdbc hsqldb /path/to/database/files

# Add a platform named 'p2' backed by a remote Oracle database
$ ixmp platform add p2 jdbc oracle \
       database.server.example.com:PORT:SCHEMA username password

# Add a platform named 'p3' with specific JVM arguments
$ ixmp platform add p3 jdbc hsqldb /path/to/database/files -Xmx12G

# Make 'p2' the default Platform
$ ixmp platform add default p2
```

…or, use the methods of *ixmp.config*.

ixmp.**config**

An instance of *Config*.

class ixmp._config.**Config**(*read: bool = True*)

Configuration for ixmp.

For most purposes, there is only one instance of this class, available at *ixmp.config* and automatically *read()* from the ixmp configuration file at the moment the package is imported. (*save()* writes the current values to file.)

Config is a key-value store. Key names are strings; each key has values of a fixed type. Individual keys can be accessed with *get()* and *set()*, or by accessing the *values* attribute.

Spaces in names are automatically replaced with underscores, e.g. "my key" is stored as "my_key", but may be set and retrieved as "my key".

Downstream packages (e.g. `message_ix`, `message_ix_models`) may *register()* additional keys to be stored in and read from the ixmp configuration file.

The default configuration (restored by *clear()*) is:

```
{
  "platform": {
    "default": "local",
    "local": {
      "class": "jdbc",
      "driver": "hsqldb",
      "path": "~/.local/share/ixmp/localdb/default"
    },
}
```

clear()	Clear all configuration keys by setting empty or default values.
get(name)	Return the value of a configuration key *name*.
keys()	Return the names of all registered configuration keys.
read()	Try to read configuration keys from file.
save()	Write configuration keys to file.
set(name, value[, _strict])	Set configuration key *name* to *value*.
register(name, type_[, default])	Register a new configuration key.
unregister(name)	Unregister and clear the configuration key *name*.
add_platform(name, *args, **kwargs)	Add or overwrite information about a platform.
get_platform_info(name)	Return information on configured Platform *name*.
remove_platform(name)	Remove the configuration for platform *name*.

Parameters

read (*bool*) – Read `config.json` on startup.

add_platform(*name: str*, **args*, ***kwargs*)

Add or overwrite information about a platform.

Parameters

- **name** (*str*) – New or existing platform name.
- **args** – Positional arguments. If *name* is 'default', *args* must be a single string: the name of an existing configured Platform. Otherwise, the first of *args* specifies one of the *BACKENDS*, and the remaining *args* differ according to the backend.
- **kwargs** – Keyword arguments. These differ according to backend.

See also:

Backend.handle_config, *JDBCBackend.handle_config*

clear()

Clear all configuration keys by setting empty or default values.

get(*name: str*) → Any

Return the value of a configuration key *name*.

get_platform_info(*name: str*) → Tuple[str, Dict[str, Any]]

Return information on configured Platform *name*.

Parameters

name (str) – Existing platform. If *name* is "default", the information for the default platform is returned.

Returns

- str – The name of the platform. If *name* was "default", this will be the actual name of platform that is designated default.
- dict – The "class" key specifies one of the *BACKENDS*. Other keys vary by backend class.

Raises

ValueError – If *name* is not configured as a platform.

keys() → Tuple[str, ...]

Return the names of all registered configuration keys.

path: Path | None = None

Fully-resolved path of the `config.json` file.

read()

Try to read configuration keys from file.

If successful, the attribute *path* is set to the path of the file.

register(*name: str*, *type_: type*, *default: Any | None = None*, ***kwargs*)

Register a new configuration key.

Parameters

- **name** (str) – Name of the new key.
- **type** (object) – Type of valid values for the key, e.g. str or pathlib.Path.
- **default** (*optional*) – Default value for the key. If not supplied, the *type* is called to supply the default value, e.g. `str()`.

Raises

ValueError – if the key *name* is already registered.

remove_platform(*name: str*)

Remove the configuration for platform *name*.

save()

Write configuration keys to file.

`config.json` is created in the first of the ixmp configuration directories that exists. Only non-null values are written.

set(*name: str*, *value: Any*, *_strict: bool = True*)

Set configuration key *name* to *value*.

Parameters

value – Value to store. If None, *set()* has no effect.

unregister(*name: str*) → None

Unregister and clear the configuration key *name*.

values: *BaseValues*

Configuration values. These can be accessed using Python item access syntax, e.g. `ixmp.config.values["platform"]["platform name"]...`.

class ixmp._config.**BaseValues**(*platform: dict* = <*factory*>)

Base class for storing configuration values.

get_field(*name*)

For *name* = "field name", retrieve a field "field_name", if any.

munge(*name*)

Return a field name matching *name*.

2.1.5 Utilities

diff(a, b[, filters])	Compute the difference between Scenarios *a* and *b*.
discard_on_error(ts)	Context manager to discard changes to *ts* and close the DB on any exception.
format_scenario_list(platform[, model, ...])	Return a formatted list of TimeSeries on *platform*.
maybe_check_out(timeseries[, state])	Check out *timeseries* depending on *state*.
maybe_commit(timeseries, condition, message)	Commit *timeseries* with *message* if *condition* is `True`.
parse_url(url)	Parse *url* and return Platform and Scenario information.
show_versions([file])	Print information about ixmp and its dependencies to *file*.
update_par(scenario, name, data)	Update parameter *name* in *scenario* using *data*, without overwriting.
to_iamc_layout(df)	Transform *df* to the IAMC structure/layout.

class ixmp.util.**DeprecatedPathFinder**(*package: str*, *name_map: Mapping[str, str]*)

Handle imports from deprecated module locations.

ixmp.util.**diff**(*a*, *b*, *filters=None*) → Iterator[Tuple[str, DataFrame]]

Compute the difference between Scenarios *a* and *b*.

diff() combines `pandas.merge()` and *Scenario.items()*. Only parameters are compared. `merge()` is called with the arguments `how="outer", sort=True, suffixes=("_a", "_b"), indicator=True`; the merge is performed on all columns except 'value' or 'unit'.

Yields

`tuple` of `str`, `pandas.DataFrame` – Tuples of item name and data.

ixmp.util.**discard_on_error**(*ts: TimeSeries*)

Context manager to discard changes to *ts* and close the DB on any exception.

For *JDBCBackend*, this can avoid leaving *ts* in a "locked" state in the database.

Examples

```
>>> mp = ixmp.Platform()
>>> s = ixmp.Scenario(mp, ...)
>>> with discard_on_error(s):
...     s.add_par(...)  # Any code
...     s.not_a_method()  # Code that raises some exception
```

Before the the exception in the final line is raised (and possibly handled by surrounding code):

- Any changes—for example, here changes due to the call to *add_par()*—are discarded/not committed;
- `s` is guaranteed to be in a non-locked state; and
- *close_db()* is called on `mp`.

ixmp.util.**format_scenario_list**(*platform*, *model=None*, *scenario=None*, *match=None*, *default_only=False*, *as_url=False*)

Return a formatted list of TimeSeries on *platform*.

Parameters

- **platform** (*Platform*) –
- **model** (`str`, *optional*) – Model name to restrict results. Passed to *scenario_list()*.
- **scenario** (`str`, *optional*) – Scenario name to restrict results. Passed to *scenario_list()*.
- **match** (`str`, *optional*) – Regular expression to restrict results. Only results where the model or scenario name matches are returned.
- **default_only** (`bool`, *optional*) – Only return TimeSeries where a default version has been set with *TimeSeries.set_as_default()*.
- **as_url** (`bool`, *optional*) – Format results as ixmp URLs.

Returns

If *as_url* is `False`, also include summary information.

Return type

`list` of `str`

ixmp.util.**logger**()

Access global logger.

Deprecated since version 3.3: To control logging from ixmp, instead use `logging` to retrieve it:

```
import logging
ixmp_logger = logging.getLogger("ixmp")

# Example: set the level to INFO
ixmp_logger.setLevel(logging.INFO)
```

ixmp.util.**maybe_check_out**(*timeseries*, *state=None*)

Check out *timeseries* depending on *state*.

If *state* is `None`, then *TimeSeries.check_out()* is called.

Returns

- `True` – if *state* was `None` and a check out was performed, i.e. *timeseries* was previously in a checked-in state.

- `False` – if *state* was `None` and no check out was performed, i.e. *timeseries* was already in a checked-out state.
- *state* – if *state* was not `None` and no check out was attempted.

Raises

ValueError – If *timeseries* is a *Scenario* object and *has_solution()* is `True`.

See also:

TimeSeries.check_out(), *Scenario.check_out()*

ixmp.util.**maybe_commit**(*timeseries*, *condition*, *message*)

Commit *timeseries* with *message* if *condition* is `True`.

Returns

- `True` – if a commit is performed.
- `False` – if any exception is raised during the attempted commit. The exception is logged with level `INFO`.

See also:

TimeSeries.commit()

ixmp.util.**maybe_convert_scalar**(*obj*) → DataFrame

Convert *obj* to `pandas.DataFrame`.

Parameters

obj – Any value returned by *Scenario.par()*. For a scalar (0-dimensional) parameter, this will be `dict`.

Returns

maybe_convert_scalar() always returns a data frame.

Return type

`pandas.DataFrame`

ixmp.util.**parse_url**(*url*)

Parse *url* and return Platform and Scenario information.

A URL (Uniform Resource Locator), as the name implies, uniquely identifies a specific scenario and (optionally) version of a model, as well as (optionally) the database in which it is stored. ixmp URLs take forms like:

```
ixmp://PLATFORM/MODEL/SCENARIO[#VERSION]
MODEL/SCENARIO[#VERSION]
```

where:

- The `PLATFORM` is a configured platform name; see *ixmp.config*.
- `MODEL` may not contain the forward slash character ('/'); `SCENARIO` may contain any number of forward slashes. Both must be supplied.
- `VERSION` is optional but, if supplied, must be an integer.

Returns

- **platform_info** (`dict`) – Keyword argument 'name' for the *Platform* constructor.
- **scenario_info** (`dict`) – Keyword arguments for a *Scenario* on the above platform: 'model', 'scenario' and, optionally, 'version'.

Raises
ValueError – For malformed URLs.

ixmp.util.**show_versions**(*file=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='utf-8'>*)

Print information about ixmp and its dependencies to *file*.

ixmp.util.**to_iamc_layout**(*df: DataFrame*) → DataFrame

Transform *df* to the IAMC structure/layout.

The returned object has:

- Any (Multi)Index levels reset as columns.
- Lower-case column names 'region', 'variable', 'subannual', and 'unit'.
- If not present in *df*, the value 'Year' in the 'subannual' column.

Parameters
df (pandas.DataFrame) – May have a 'node' column, which will be renamed to 'region'.

Return type
pandas.DataFrame

Raises
ValueError – If 'region', 'variable', or 'unit' is not among the column names.

ixmp.util.**update_par**(*scenario*, *name*, *data*)

Update parameter *name* in *scenario* using *data*, without overwriting.

Only values which do not already appear in the parameter data are added.

Utilities for documentation

GitHub adapter for sphinx.ext.linkcode.

To use this extension, add it to the `extensions` setting in the Sphinx configuration file (usually `conf.py`), and set the required `linkcode_github_repo_slug`:

```
extensions = [
    ...,
    "ixmp.util.sphinx_linkcode_github",
    ...,
]

linkcode_github_repo_slug = "iiasa/ixmp"  # Required
linkcode_github_remote_head = "feature/example"  # Optional
```

The extension uses GitPython (if installed) or `linkcode_github_remote_head` (optional override) to match a local commit to a remote head (~branch name), and construct links like:

```
https://github.com/{repo_slug}/blob/{remote_head}/path/to/source.py#L123-L456
```

class ixmp.util.sphinx_linkcode_github.**GitHubLinker**

Handler for storing files/line numbers for code objects and formatting links.

autodoc_process_docstring(*app: sphinx.application.Sphinx*, *what*, *name: str*, *obj*, *options*, *lines*)

Handler for the Sphinx `autodoc-process-docstring` event.

Records the file and source line numbers containing *obj*.

config_inited(*app: sphinx.application.Sphinx*, *config*)

Handler for the Sphinx `config-inited` event.

linkcode_resolve(*domain: str*, *info: dict*) → str | None

Function for the `sphinx.ext.linkcode` setting of the same name.

Returns URLs for code objects on GitHub, using information stored by `autodoc_process_docstring()`.

ixmp.util.sphinx_linkcode_github.**find_remote_head**(*app: sphinx.application.Sphinx*) → str

Return a name for the remote branch containing the code.

ixmp.util.sphinx_linkcode_github.**find_remote_head_git**(*app: sphinx.application.Sphinx*) → str | None

Use git to identify the name of the remote branch containing the code.

ixmp.util.sphinx_linkcode_github.**package_base_path**(*obj*) → Path

Return the base path of the package containing *obj*.

ixmp.util.sphinx_linkcode_github.**setup**(*app: sphinx.application.Sphinx*)

Sphinx extension registration hook.

Utilities for testing

Utilities for testing ixmp.

These include:

- pytest hooks, fixtures:

ixmp_cli	A CliRunner object that invokes the ixmp command-line interface.
tmp_env	Return the os.environ dict with the IXMP_DATA variable set.
test_mp	An empty *Platform* connected to a temporary, in-memory database.

…and assertions:

assert_logs(caplog[, message_or_messages, ...])	Assert that *message_or_messages* appear in logs.

- Methods for setting up and populating test ixmp databases:

add_test_data(scen)	Populate *scen* with test data.
create_test_platform(tmp_path, data_path, ...)	Create a Platform for testing using specimen files '*name*.*'.
make_dantzig(mp[, solve, quiet])	Return *ixmp.Scenario* of Dantzig's canning/transport problem.
populate_test_platform(platform)	Populate *platform* with data for testing.

- Methods to run and retrieve values from Jupyter notebooks:

run_notebook(nb_path, tmp_path[, env])	Execute a Jupyter notebook via `nbclient` and collect output.
get_cell_output(nb, name_or_index[, kind])	Retrieve a cell from *nb* according to its metadata *name_or_index*:

`ixmp.testing.`**`add_random_model_data`**(*scenario*, *length*)

Add a set and parameter with given *length* to *scenario*.

The set is named 'random_set'. The parameter is named 'random_par', and has two dimensions indexed by 'random_set'.

`ixmp.testing.`**`add_test_data`**(*scen:* Scenario)

Populate *scen* with test data.

`ixmp.testing.`**`assert_logs`**(*caplog*, *message_or_messages=None*, *at_level=None*)

Assert that *message_or_messages* appear in logs.

Use assert_logs as a context manager for a statement that is expected to trigger certain log messages. assert_logs checks that these messages are generated.

Example

def test_foo(caplog):

with assert_logs(caplog, 'a message'):
logging.getLogger(__name__).info('this is a message!')

Parameters

- **`caplog`** (`object`) – The pytest caplog fixture.
- **`message_or_messages`** (`str` or `list` of `str`) – String(s) that must appear in log messages.
- **`at_level`** (`int`, *optional*) – Messages must appear on 'ixmp' or a sub-logger with at least this level.

`ixmp.testing.`**`create_test_platform`**(*tmp_path*, *data_path*, *name*, ***properties*)

Create a Platform for testing using specimen files '*name*.*'.

Any of the following files from *data_path* are copied to *tmp_path*:

- *name*.lobs, *name*.script, i.e. the contents of a *JDBCBackend* HyperSQL database.
- *name*.properties.

The contents of *name*.properties (if it exists) are formatted using the *properties* keyword arguments.

Returns
the path to the .properties file, if any, else the .lobs file without suffix.

Return type
`pathlib.Path`

ixmp.testing.**get_cell_output**(*nb*, *name_or_index*, *kind='data'*)

Retrieve a cell from *nb* according to its metadata *name_or_index*:

The Jupyter notebook format allows specifying a document-wide unique 'name' metadata attribute for each cell:

https://nbformat.readthedocs.io/en/latest/format_description.html #cell-metadata

Return the cell matching *name_or_index* if `str`; or the cell at the `int` index; or raise `ValueError`.

Parameters

kind (`str`, *optional*) – Kind of cell output to retrieve. For 'data', the data in format 'text/plain' is run through `eval()`. To retrieve an exception message, use 'evalue'.

ixmp.testing.**ixmp_cli**(*tmp_env*)

A CliRunner object that invokes the ixmp command-line interface.

ixmp.testing.**make_dantzig**(*mp:* Platform, *solve:* *bool* = *False*, *quiet:* *bool* = *False*) → *Scenario*

Return *ixmp.Scenario* of Dantzig's canning/transport problem.

Parameters

- **mp** (*Platform*) – Platform on which to create the scenario.
- **solve** (`bool`, *optional*) – If `True`. then solve the scenario before returning. Default `False`.
- **quiet** (`bool`, *optional*) – If `True`, suppress console output when solving.

Return type

Scenario

See also:

DantzigModel

ixmp.testing.**populate_test_platform**(*platform*)

Populate *platform* with data for testing.

Many of the tests in `ixmp.tests.core` depend on this set of data.

The data consist of:

- 3 versions of the Dantzig cannery/transport Scenario.
  - Version 2 is the default.
  - All have *HIST_DF* and *TS_DF* as time-series data.
- 1 version of a TimeSeries with model name 'Douglas Adams' and scenario name 'Hitchhiker', containing 2 values.

ixmp.testing.**random_model_data**(*length*)

Random (set, parameter) data with at least *length* elements.

See also:

add_random_model_data

ixmp.testing.**random_ts_data**(*length*)

A `pandas.DataFrame` of time series data with *length* rows.

Suitable for passage to *TimeSeries.add_timeseries()*.

ixmp.testing.**resource_limit**(*request*)

A fixture that limits Python `resources`.

See the documentation (`pytest --help`) for the `--resource-limit` command-line option that selects (1) the specific resource and (2) the level of the limit.

The original limit, if any, is reset after the test function in which the fixture is used.

ixmp.testing.**run_notebook**(*nb_path*, *tmp_path*, *env=None*, ***kwargs*)

Execute a Jupyter notebook via `nbclient` and collect output.

Parameters

- **nb_path** (`os.PathLike`) – The notebook file to execute.
- **tmp_path** (`os.PathLike`) – A directory in which to create temporary output.
- **env** (`dict`, *optional*) – Execution environment for `nbclient`. Default: `os.environ`.
- **kwargs** – Keyword arguments for `nbclient.NotebookClient`. Defaults are set for:

  "allow_errors"
  Default `False`. If `True`, the execution always succeeds, and cell output contains exception information rather than code outputs.

  "kernel_version"
  Jupyter kernel to use. Default: either "python2" or "python3", matching the current Python major version.

  > **Warning:** Any existing configuration for this kernel on the local system— such as an IPython start-up file—will be executed when the kernel starts. Code that enables GUI features can interfere with *run_notebook()*.

  "timeout"
  in seconds; default 10.

Returns

- **nb** (`nbformat.NotebookNode`) – Parsed and executed notebook.
- **errors** (`list`) – Any execution errors.

ixmp.testing.**test_mp**(*request*, *tmp_env*, *test_data_path*)

An empty *Platform* connected to a temporary, in-memory database.

This fixture has **module** scope: the same Platform is reused for all tests in a module.

ixmp.testing.**tmp_env**(*pytestconfig*, *tmp_path_factory*)

Return the os.environ dict with the IXMP_DATA variable set.

IXMP_DATA will point to a temporary directory that is unique to the test session. ixmp configuration (i.e. the 'config.json' file) can be written and read in this directory without modifying the current user's configuration.

ixmp.testing.data.**HIST_DF = model scenario region variable unit 2000 2005 2010 0 canning problem standard DantzigLand GDP USD 850.0 900.0 950.0**

Time series data for testing.

ixmp.testing.data.**TS_DF = model scenario region variable unit 2000 2005 2010 0 canning problem standard DantzigLand Demand cases 850.0 900.0 NaN 1 canning problem standard DantzigLand GDP USD 850.0 900.0 950.0**

Time series data for testing.

2.2 Usage in R via `reticulate`

ixmp is fully usable in R via reticulate, a package that allows nearly seamless access to the Python API. No additional R packages are needed.

Note: The former `rixmp` package was removed in *ixmp v3.3.0 (2021-05-28)*.

See Install R and reticulate for installing R and reticulate to use with *ixmp*. Those instructions are suitable whether `message_ix` is also installed, or only *ixmp*.

Once installed, use reticulate to import the Python package:

```
library(reticulate)
ixmp <- import("ixmp")
```

This creates a global variable, `ixmp`, that can be used much like the Python module:

```
mp <- ixmp$Platform(name = 'default')
scen <- ixmp$Scenario(mp, 'model name', 'scenario name', version = 'new')
```

Finally, see the R versions of the *Tutorials*.

2.3 Storage back ends (`ixmp.backend`)

ixmp includes *ixmp.backend.jdbc.JDBCBackend*, which can store data in many types of relational database management systems (RDBMS) that have Java DataBase Connector (JDBC) interfaces—hence its name.

ixmp is extensible to support other methods of storing data: in non-JDBC RDBMS, non-relational databases, local files, memory, or other ways. Developers wishing to add such capabilities may subclass *ixmp.backend.base.Backend* and implement its methods.

2.3.1 Provided backends

ixmp.backend.BACKENDS: Dict[str, Type] = {'jdbc': <class 'ixmp.backend.jdbc.JDBCBackend'>}

Mapping from names to available backends. To register additional backends, add entries to this dictionary.

class ixmp.backend.jdbc.**JDBCBackend**(*jvmargs=None*, ***kwargs*)

Backend using JPype/JDBC to connect to Oracle and HyperSQL databases.

This backend is based on the third-party JPype Python package that allows interaction with Java code.

Parameters

- **driver** (`'oracle'` or `'hsqldb'`) – JDBC driver to use.
- **path** (`os.PathLike`, *optional*) – Path to the HyperSQL database.

- **url** (str, *optional*) – Partial or complete JDBC URL for the Oracle or HyperSQL database, e.g. `database-server.example.com:PORT:SCHEMA`. See *Configuration*.
- **user** (str, *optional*) – Database user name.
- **password** (str, *optional*) – Database user password.
- **cache** (bool, *optional*) – If `True` (the default), cache Python objects after conversion from Java objects.
- **jvmargs** (str, *optional*) – Java Virtual Machine arguments. See *start_jvm()*.
- **dbprops** (os.PathLike, *optional*) – With `driver='oracle'`, the path to a database properties file containing *driver*, *url*, *user*, and *password* information.

JDBCBackend supports:

- Databases in local files (HyperSQL) using `driver='hsqldb'` and the *path* argument.
- Remote, Oracle databases using `driver='oracle'` and the *url*, *username* and *password* arguments.
- Temporary, in-memory databases using `driver='hsqldb'` and the *url* argument. Use the *url* parameter with the format `jdbc:hsqldb:mem:[NAME]`, where [NAME] is any string:

```
mp = ixmp.Platform(
    backend="jdbc",
    driver="hsqldb",
    url="jdbc:hsqldb:mem:temporary platform",
)
```

JDBCBackend caches values in memory to improve performance when repeatedly reading data from the same items with *par()*, *equ()*, or *var()*.

Tip: If repeatedly accessing the same item with different *filters*:

1. First, access the item by calling e.g. *par()* *without* any filters. This causes the full contents of the item to be loaded into cache.
2. Then, access by making multiple *par()* calls with different *filters* arguments. The cache value is filtered and returned without further access to the database.

Tip: Modifying an item by adding or deleting elements invalidates its cache.

JDBCBackend has the following **limitations**:

- The *comment* argument to *Platform.add_unit()* is limited to 64 characters.
- Infinite floating-point values (numpy.inf, math.inf) cannot be stored using *TimeSeries.add_timeseries()* when using an Oracle database via `driver='oracle'`.

JDBCBackend's implementation allows the following kinds of file input and output:

read_file(path, item_type, **kwargs)	Read Platform, TimeSeries, or Scenario data from file.
write_file(path, item_type, **kwargs)	Write Platform, TimeSeries, or Scenario data to file.

classmethod handle_config(*args*, *kwargs*)

Handle platform/backend config arguments.

args will overwrite any *kwargs*, and may be one of:

- ("oracle", url, user, password, [jvmargs]) for an Oracle database.
- ("hsqldb", path, [jvmargs]) for a file-backed HyperSQL database.
- ("hsqldb",) with "url" supplied via *kwargs*, e.g. "jdbc:hsqldb:mem://foo" for an in-memory database.

read_file(*path*, *item_type:* ItemType, ***kwargs*)

Read Platform, TimeSeries, or Scenario data from file.

JDBCBackend supports reading from:

- `path='*.gdx', item_type=ItemType.MODEL`. The keyword arguments *check_solution*, *comment*, *equ_list*, and *var_list* are **required**.

  Parameters

  - **check_solution** (`bool`) – If True, raise an exception if the GAMS solver did not reach optimality. (Only for MESSAGE-scheme Scenarios.)
  - **comment** (`str`) – Comment added to Scenario when importing the solution.
  - **equ_list** (`list` of `str`) – Equations to be imported.
  - **var_list** (`list` of `str`) – Variables to be imported.
  - **filters** (`dict` of `dict` of `str`) – Restrict items read.

See also:

Backend.read_file

write_file(*path*, *item_type:* ItemType, ***kwargs*)

Write Platform, TimeSeries, or Scenario data to file.

JDBCBackend supports writing to:

- `path='*.gdx', item_type=ItemType.SET | ItemType.PAR`.
- `path='*.csv', item_type=TS`. The *default* keyword argument is **required**.

  Parameters

  filters (`dict` of `dict` of `str`) – Restrict items written. The following filters may be used:

  - model : str
  - scenario : str
  - variable : list of str
  - default : bool. If `True`, only data from TimeSeries versions with *TimeSeries.set_as_default()* are written.

See also:

Backend.write_file

ixmp.backend.jdbc.**start_jvm**(*jvmargs=None*)

Start the Java Virtual Machine via JPype.

Parameters

jvmargs (str or list of str, *optional*) – Additional arguments for launching the JVM, passed to jpype.startJVM().

For instance, to set the maximum heap space to 4 GiB, give jvmargs=['-Xmx4G']. See the JVM documentation for a list of options.

2.3.2 Backend API

ixmp.backend.base.Backend()	Abstract base class for backends.
ixmp.backend.base.CachingBackend([cache_enabled])	Backend with additional features for caching data.
ixmp.backend.ItemType(value[, names, ...])	Type of data items in *TimeSeries* and *Scenario*.
ixmp.backend.FIELDS	Lists of field names for tuples returned by Backend API methods.
ixmp.backend.IAMC_IDX	Partial list of dimensions for the IAMC data structure, or "IAMC format".

- *ixmp.Platform* implements a *user-friendly* API for scientific programming. This means its methods can take many types of arguments, check, and transform them—in a way that provides modeler-users with easy, intuitive workflows.
- In contrast, *Backend* has a *very simple* API that accepts arguments and returns values in basic Python data types and structures.
- As a result:
  - *Platform* code is not affected by where and how data is stored; it merely handles user arguments and then makes, usually, a single *Backend* call.
  - *Backend* code does not need to perform argument checking; merely store and retrieve data reliably.
- Additional Backends may inherit from *Backend* or *CachingBackend*.

class ixmp.backend.base.**Backend**

Abstract base class for backends.

In the following, the bold-face words **required**, **optional**, etc. have specific meanings as described in IETF RFC 2119.

Backend is an **abstract** class; this means it **must** be subclassed. Most of its methods are decorated with abc.abstractmethod; this means they are **required** and **must** be overridden by subclasses.

Others, marked below with "OPTIONAL:", are not so decorated. For these methods, the behaviour in the base Backend—often, nothing—is an acceptable default behaviour. Subclasses **may** extend or replace this behaviour as desired, so long as the methods still perform the actions described in the description.

Backends:

- **must** only raise standard Python exceptions.
- **must** implement the *data model* as described, or raise NotImplementedError for not implemented parts of the data model.

Methods related to *ixmp.Platform*:

add_model_name	Add (register) new model name.
add_scenario_name	Add (register) new scenario name.
close_db	OPTIONAL: Close database connection(s).
get_auth	OPTIONAL: Return user authorization for *models*.
get_doc	Read documentation from database
get_log_level	OPTIONAL: Get logging level for the backend and other code.
get_meta	Retrieve all metadata attached to a specific target.
get_model_names	List existing model names.
get_nodes	Iterate over all nodes stored on the Platform.
get_scenarios	Iterate over TimeSeries stored on the Platform.
get_scenario_names	List existing scenario names.
get_units	Return all registered symbols for units of measurement.
handle_config	OPTIONAL: Handle platform/backend config arguments.
open_db	OPTIONAL: (Re-)open database connection(s).
read_file	OPTIONAL: Read Platform, TimeSeries, or Scenario data from file.
remove_meta	Remove metadata attached to a target.
set_doc	Save documentation to database
set_log_level	OPTIONAL: Set logging level for the backend and other code.
set_meta	Set metadata on a target.
set_node	Add a node name to the Platform.
set_unit	Add a unit of measurement to the Platform.
write_file	OPTIONAL: Write Platform, TimeSeries, or Scenario data to file.

Methods related to *ixmp.TimeSeries*:

- Each method has an argument *ts*, a reference to the TimeSeries object being manipulated.
- 'Geodata' is otherwise identical to regular timeseries data, except value are `str` rather than `float`.

check_out	Check out *ts* for modification.
commit	Commit changes to *ts*.
delete	Remove time series data.
delete_geo	Remove 'geodata' values.
discard_changes	Discard changes to *ts* since the last *check_out()*.
get	Retrieve the existing TimeSeries (or Scenario) *ts*.
get_data	Retrieve time series data.
get_geo	Retrieve time-series 'geodata'.
init	Create a new TimeSeries (or Scenario) *ts*.
is_default	Return `True` if *ts* is the default version for its (model, scenario).
last_update	Return the date of the last modification of the *ts*, if any.
preload	OPTIONAL: Load *ts* data into memory.
run_id	Return the run ID of the *ts*.
set_data	Store *data*.
set_as_default	Set the current *TimeSeries.version* as the default.
set_geo	Store time series geodata.

Methods related to *ixmp.Scenario*:

- Each method has an argument *s*, a reference to the Scenario object being manipulated.

clone	Clone *s*.
delete_item	Remove an item *name* of *type*.
get_meta	Retrieve all metadata attached to a specific target.
has_solution	Return *True* if Scenario *s* has been solved.
init_item	Initialize an item *name* of *type*.
item_delete_elements	Remove elements of item *name*.
item_get_elements	Return elements of item *name*.
item_set_elements	Add keys or values to item *name*.
item_index	Return the index sets or names of item *name*.
list_items	Return a list of names of items of *type*.
remove_meta	Remove metadata attached to a target.
set_meta	Set metadata on a target.

Methods related to `message_ix.Scenario`:

- Each method has an argument *ms*, a reference to the Scenario object being manipulated.

> **Warning:** These methods may be moved to ixmp in a future release.

cat_get_elements	Get elements of a category mapping.
cat_list	Return list of categories in mapping *name*.
cat_set_elements	Add elements to category mapping.

abstract add_model_name(*name: str*) → None

Add (register) new model name.

Parameters
name (str) – New model name

abstract add_scenario_name(*name: str*) → None

Add (register) new scenario name.

Parameters
name (str) – New scenario name

abstract cat_get_elements(*ms:* Scenario, *name: str*, *cat: str*) → List[str]

Get elements of a category mapping.

Parameters

- **name** (str) – Name of the category mapping set.
- **cat** (str) – Name of the category within *name*.

Returns
All set elements mapped to *cat* in *name*.

Return type
list of str

abstract cat_list(*ms:* Scenario, *name: str*) → List[str]

Return list of categories in mapping *name*.

Parameters
name (str) – Name of the category mapping set.

Returns
All categories in *name*.

Return type
list of str

abstract cat_set_elements(*ms:* Scenario, *name: str*, *cat: str*, *keys: str | Sequence[str]*, *is_unique: bool*) → None

Add elements to category mapping.

Parameters

- **name** (str) – Name of the category mapping set.
- **cat** (str) – Name of the category within *name*.
- **keys** (collections.abc.Iterable of str or list of str) – Keys to add to *cat*.
- **is_unique** (bool) – If True:
  - *keys* **must** contain only one key.
  - The Backend **must** remove any existing member of *cat*, so that it has only one element.

abstract check_out(*ts:* TimeSeries, *timeseries_only: bool*) → None

Check out *ts* for modification.

Parameters
timeseries_only (bool) – ???

abstract clear_solution(*s:* Scenario, *from_year=None*)

Remove data associated with a model solution.

Todo: Document.

---

abstract clone(*s:* Scenario, *platform_dest:* Platform, *model: str*, *scenario: str*, *annotation: str*, *keep_solution: bool*, *first_model_year: int | None = None*) → *Scenario*

Clone *s*.

Parameters

- **platform_dest** (*Platform*) – Target backend. May be the same as s.platform.
- **model** (str) – New model name.
- **scenario** (str) – New scenario name.
- **annotation** (str) – Description for the creation of the new scenario.
- **keep_solution** (bool) – If True, model solution data is also cloned. If False, it is discarded.
- **first_model_year** (int or None) – If int, must be greater than the first model year of *s*.

Returns
The cloned Scenario. If *s* is an instance of a subclass of *ixmp.Scenario*, the returned object **must** be of the same subclass.

Return type
Scenario

close_db() → None

OPTIONAL: Close database connection(s).

Close any database connection(s), if open.

abstract commit(*ts:* TimeSeries, *comment: str*) → None

Commit changes to *ts*.

ts_init **may** modify the *version* attribute of *ts*.

Parameters
comment (str) – Description of the changes being committed.

del_ts(*ts:* TimeSeries) → None

OPTIONAL: Free memory associated with the TimeSeries *ts*.

The default implementation has no effect.

abstract delete(*ts:* TimeSeries, *region: str*, *variable: str*, *subannual: str*, *years: Iterable[int]*, *unit: str*) → None

Remove time series data.

Parameters

- **region** (str) – Region name.
- **variable** (str) – Variable name.
- **years** (Iterable of int) – Years.
- **unit** (str) – Unit symbol.
- **subannual** (str) – Name of time slice.

abstract delete_geo(*ts:* TimeSeries, *region: str*, *variable: str*, *subannual: str*, *years: Iterable[int]*, *unit: str*) → None

Remove 'geodata' values.

Parameters

- **region** (str) – Region name.
- **variable** (str) – Variable name.
- **subannual** (str) – Name of time slice.
- **years** (Iterable of int) – Years.
- **unit** (str) – Unit symbol.

abstract delete_item(*s:* Scenario, *type: str*, *name: str*) → None

Remove an item *name* of *type*.

Parameters

- **type** ('set' or 'par' or 'equ') –
- **name** (str) – Name of the item to delete.

abstract discard_changes(*ts:* TimeSeries) → None

Discard changes to *ts* since the last *check_out()*.

abstract get(*ts:* TimeSeries) → None

Retrieve the existing TimeSeries (or Scenario) *ts*.

The TimeSeries is identified based on the unique combination of the attributes of *ts*:

- *model*,
- *scenario*, and
- *version*.

If *version* is None, the Backend **must** return the version marked as default, and **must** set the attribute value.

If *ts* is a Scenario, *get()* **must** set the *scheme* attribute with the value previously passed to *init()*.

Raises

ValueError – If *model* or *scenario* does not exist on the Platform.

See also:

is_default, *set_as_default*

get_auth(*user: str*, *models: Sequence[str]*, *kind: str*) → Dict[str, bool]

OPTIONAL: Return user authorization for *models*.

If the Backend implements access control, this method **must** indicate whether *user* has permission *kind* for each of *models*.

kind **may** be 'read'/'view', 'write'/'modify', or other values; *get_auth()* **should** raise exceptions on invalid values.

Parameters

- **user** (str) – User name or identifier.
- **models** (list of str) – Model names.
- **kind** (str) – Type of permission being requested

Returns

Mapping of *model name (str)* → bool; True if the user is authorized for the model.

Return type

dict

abstract get_data(*ts:* TimeSeries, *region: Sequence[str]*, *variable: Sequence[str]*, *unit: Sequence[str]*, *year: Sequence[str]*) → Iterable[Tuple[str, str, str, int, float]]

Retrieve time series data.

Parameters

- **region** (list of str) – Region names to filter results.
- **variable** (list of str) – Variable names to filter results.
- **unit** (list of str) – Unit symbols to filter results.
- **year** (list of str) – Years to filter results.

Yields

tuple – The members of each tuple are:

ID	Type	Description
region	str	Region name
variable	str	Variable name
unit	str	Unit symbol
year	int	Year
value	float	Data value

abstract get_doc(*domain: str*, *name: str | None = None*) → str | Dict

Read documentation from database

Parameters

- **domain** (str) – Documentation domain, e.g. model, scenario etc
- **name** (str, *optional*) – Name of domain entity (e.g. model name).

Returns

String representing fragment of documentation if name is passed as parameter or dictionary containing mapping between name of domain object (e.g. model name) and string representing fragment when name parameter is omitted.

Return type

str or dict

abstract get_geo(*ts: TimeSeries*) → Iterable[Tuple[str, str, int, str, str, str, bool]]

Retrieve time-series 'geodata'.

Yields

tuple – The members of each tuple are:

ID	Type	Description
region	str	Region name
variable	str	Variable name
year	int	Year
value	str	Value
unit	str	Unit symbol
subannual	str	Name of time slice
meta	bool	True if the data is marked as metadata

get_log_level() → str

OPTIONAL: Get logging level for the backend and other code.

The default implementation returns the effective level of the "ixmp.backend.base" logger; usually the same as "ixmp" or "ixmp.backend" (if set).

Returns

Name of a Python logging level.

Return type

str

See also:

set_log_level

abstract get_meta(*model: str | None*, *scenario: str | None*, *version: int | None*, *strict: bool*) → Dict[str, Any]

Retrieve all metadata attached to a specific target.

Depending on which of *model*, *scenario*, *version* are `None`, metadata attached to one of the four kinds of metadata targets (see *Metadata*) is returned.

If *strict* is `False`, then *get_meta()* **must** also return metadata attached to less specific or "higher level" targets:

- For (model, scenario, version), these are (model, scenario); (model,); and (scenario).
- For (model, scenario), these are (model,) and (scenario,).
- For (model,) or (scenario,), there are no less specific targets.

Parameters

- **model** (`str`, *optional*) – Model name of metadata target.
- **scenario** (`str`, *optional*) – Scenario name of metadata target.
- **version** (`int`, *optional*) – *TimeSeries.version* of metadata target.
- **strict** (`bool`) – Only retrieve metadata from the specified target.

Returns
Mapping from metadata names/identifiers (`str`) to values (`Any`).

Return type
`dict`

Raises
ValueError – on unsupported (*model*, *scenario*, *version*) combinations.

abstract get_model_names() → Iterable[str]

List existing model names.

Returns
List of the retrieved model names.

Return type
`list` of `str`

abstract get_nodes() → Iterable[Tuple[str, str | None, str, str]]

Iterate over all nodes stored on the Platform.

Yields
`tuple` – The members of each tuple are:

ID	Type	Description
region	str	Node name or synonym for node
mapped_to	str or None	Node name
parent	str	Parent node name
hierarchy	str	Node hierarchy ID

See also:

set_node

abstract get_scenario_names() → Iterable[str]

List existing scenario names.

Returns
List of the retrieved scenario names.

Return type
list of str

abstract get_scenarios(*default: bool*, *model: str | None*, *scenario: str | None*) → Iterable[Tuple[str, str, str, bool, bool, str, str, str, str, str, str, str, int]]

Iterate over TimeSeries stored on the Platform.

Scenarios, as subclasses of TimeSeries, are also included.

Parameters

- **default** (bool) – True to include only TimeSeries versions marked as default.
- **model** (str or None) – Model name to filter results.
- **scenario** (str or None) – Scenario name to filter results.

Yields
tuple – The members of each tuple are:

ID	Type	Description
model	str	Model name
scenario	str	Scenario name
scheme	str	Scheme name
is_default	bool	True if *version* is the default
is_locked	bool	True if read-only
cre_user	str	Name of user who created the TimeSeries
cre_date	str	Creation datetime
upd_user	str	Name of user who last modified the TimeSeries
upd_date	str	Modification datetime
lock_user	str	Name of user who locked the TimeSeries
lock_date	str	Lock datetime
annotation	str	Description of the TimeSeries
version	int	Version

abstract get_timeslices() → Iterable[Tuple[str, str, float]]

Iterate over subannual timeslices defined on the Platform instance.

Yields
tuple – The members of each tuple are:

ID	Type	Description
name	str	Time slice name
category	str	Time slice category
duration	float	Time slice duration (fraction of year)

See also:

set_timeslice

abstract get_units() → List[str]

Return all registered symbols for units of measurement.

Return type
list of str

See also:

set_unit

classmethod handle_config(*args: Sequence*, *kwargs: MutableMapping*) → Dict[str, Any]

OPTIONAL: Handle platform/backend config arguments.

Returns a dict to be stored in the configuration file. This dict **must** be valid as keyword arguments to the __init__() method of a Backend subclass.

The default implementation expects both *args* and *kwargs* to be empty.

See also:

Config.add_platform

abstract has_solution(*s:* Scenario) → bool

Return *True* if Scenario *s* has been solved.

If True, model solution data is available from the Backend.

abstract init(*ts:* TimeSeries, *annotation: str*) → None

Create a new TimeSeries (or Scenario) *ts*.

init **may** modify the *version* attribute of *ts*.

If *ts* is a *Scenario*; the Backend **must** store the *Scenario.scheme* attribute.

Parameters
annotation (str) – If *ts* is newly-created, the Backend **must** store this annotation with the TimeSeries.

abstract init_item(*s:* Scenario, *type: str*, *name: str*, *idx_sets: Sequence[str]*, *idx_names: Sequence[str] | None*) → None

Initialize an item *name* of *type*.

Parameters

- **type** ('set' or 'par' or 'equ' or 'var') –
- **name** (str) – Name for the new item.
- **idx_sets** (collections.abc.Sequence of str) – If empty, a 0-dimensional/scalar item is initialized. Otherwise, a 1+-dimensional item is initialized.
- **idx_names** (collections.abc.Sequence of str or None) – Optional names for the dimensions. If not supplied, the names of the *idx_sets* (if any) are used. If supplied, *idx_names* and *idx_sets* must be the same length.

Raises
ValueError – if any of the *idx_sets* is not an existing set in the Scenario; if *idx_names* and *idx_sets* are not the same length.

abstract is_default(*ts:* TimeSeries) → bool

Return True if *ts* is the default version for its (model, scenario).

See also:

get, *set_as_default*

abstract item_delete_elements(*s:* Scenario, *type: str*, *name: str*, *keys*) → None

Remove elements of item *name*.

Parameters

- **type** (`'par'` or `'set'`) –
- **name** (`str`) –
- **keys** (`collections.abc.Iterable` of `collections.abc.Iterable` of `str`) – If *name* is indexed by other set(s), then the number of elements of each key in *keys*, and their contents, must match the index set(s). If *name* is a basic set, then each key must be a list containing a single str, which must exist in the set.

See also:

init_item, *item_set_elements*

abstract item_get_elements(*s:* Scenario, *type: Literal['equ', 'par', 'set', 'var']*, *name: str*, *filters: Dict[str, List[Any]] | None = None*) → Dict[str, Any] | Series | DataFrame

Return elements of item *name*.

Parameters

- **type** (`str`) – Type of the item.
- **name** (`str`) – Name of the item.
- **filters** (`dict`, *optional*) – If provided, a mapping from dimension names (class:*str*) to allowed values along that dimension (`list`).

  item_get_elements **must** silently accept values that are *not* members of the set indexing a dimension. Elements which are not `str` **must** be handled as equivalent to their string representation; that is, item_get_elements must return the same data for `filters={'foo': [42]}` and `filters={'foo': ['42']}`.

Returns

- `pandas.Series` – When *type* is 'set' and *name* an index set (not indexed by other sets).
- `dict` – When *type* is 'equ', 'par', or 'var' and *name* is scalar (zero- dimensional). The value has the keys 'value' and 'unit' (for 'par') or 'lvl' and 'mrg' (for 'equ' or 'var').
- `pandas.DataFrame` – For mapping sets, or all 1+-dimensional values. The dataframe has one column per index name with dimension values; plus the columns 'value' and 'unit' (for 'par') or 'lvl' and 'mrg' (for 'equ' or 'var').

Raises

KeyError – If *name* does not exist in *s*.

abstract item_index(*s:* Scenario, *name: str*, *sets_or_names: str*) → List[str]

Return the index sets or names of item *name*.

Parameters

sets_or_names (`'sets'` or `'names'`) –

Return type

`list` of `str`

abstract item_set_elements(*s:* Scenario, *type: str*, *name: str*, *elements: Iterable[Tuple[Any, float | None, str | None, str | None]]*) → None

Add keys or values to item *name*.

Parameters

- **type** ('par' or 'set') –
- **name** (str) – Name of the items.
- **elements** (collections.abc.Iterable of tuple) – The members of each tuple are:

ID	Type	Description
key	str or list of str or None	Set elements or value indices
value	float or None	Parameter value
unit	str or None	Unit symbol
comment	str or None	Description of the change

If *name* is indexed by other set(s), then the number of elements of each *key*, and their contents, must match the index set(s). When *type* is 'set', *value* and *unit* **must** be None.

Raises

- **ValueError** – If *elements* contain invalid values, e.g. key values not in the index set(s).
- **Exception** – If the Backend encounters any error adding the elements.

See also:

init_item, *item_delete_elements*

abstract last_update(*ts:* TimeSeries) → str | None

Return the date of the last modification of the *ts*, if any.

abstract list_items(*s:* Scenario, *type: str*) → List[str]

Return a list of names of items of *type*.

Parameters
type ('set' or 'par' or 'equ') –

open_db() → None

OPTIONAL: (Re-)open database connection(s).

A backend **may** connect to a database server. This method opens the database connection if it is closed.

preload(*ts:* TimeSeries) → None

OPTIONAL: Load *ts* data into memory.

read_file(*path: PathLike*, *item_type:* ItemType, ***kwargs*) → None

OPTIONAL: Read Platform, TimeSeries, or Scenario data from file.

A backend **may** implement read_file for one or more combinations of the *path* and *item_type* methods. For all other combinations, it **must** raise NotImplementedError.

The default implementation supports:

- *path* ending in '.xlsx', *item_type* is ItemType.MODEL: read a single Scenario given by `kwargs['filters']['scenario']` from file, using *s_read_excel()*.

Parameters

- **path** (os.PathLike) – File for input. The filename suffix determines the input format:

Suffix	Format
.csv	Comma-separated values
.gdx	GAMS data exchange
.xlsx	Microsoft Office Open XML spreadsheet

- **item_type** (*ItemType*) – Type(s) of items to read.

Raises

- **ValueError** – If *ts* is not None and 'scenario' is a key in *filters*.
- **NotImplementedError** – If input of the specified items from the file format is not supported.

See also:

write_file

abstract remove_meta(*names: list*, *model: str | None*, *scenario: str | None*, *version: int | None*) → None

Remove metadata attached to a target.

Parameters

- **names** (list of str) – Metadata names/identifiers to remove.
- **model** (str or None) – Model name of metadata target.
- **scenario** (str or None) – Scenario name of metadata target.
- **version** (int or None) – *TimeSeries.version* of metadata target.

Raises

ValueError – on unsupported (*model*, *scenario*, *version*) combinations.

See also:

get_meta

abstract run_id(*ts:* TimeSeries) → int

Return the run ID of the *ts*.

abstract set_as_default(*ts:* TimeSeries) → None

Set the current *TimeSeries.version* as the default.

See also:

get, *is_default*

abstract set_data(*ts:* TimeSeries, *region: str*, *variable: str*, *data: Dict[int, float]*, *unit: str*, *subannual: str*, *meta: bool*) → None

Store *data*.

Parameters

- **region** (str) – Region name.
- **variable** (str) – Variable name.
- **subannual** (str) – Name of time slice.
- **unit** (str) – Unit symbol.
- **data** – Mapping from year (int) to value (float).

- **meta** (bool) – True to mark *data* as metadata.

abstract set_doc(*domain: str*, *docs*) → None

Save documentation to database

Parameters

- **domain** (str) – Documentation domain, e.g. model, scenario, etc.
- **docs** (dict or collections.abc.Iterable of tuple) – Dictionary or tuple array containing mapping between name of domain object (e.g. model name) and string representing fragment of documentation.

abstract set_geo(*ts: TimeSeries*, *region: str*, *variable: str*, *subannual: str*, *year: int*, *value: str*, *unit: str*, *meta: bool*) → None

Store time series geodata.

Parameters

- **region** (str) – Region name.
- **variable** (str) – Variable name.
- **subannual** (str) – Name of time slice.
- **year** (int) – Year.
- **value** (str) – Data value.
- **unit** (str) – Unit symbol.
- **meta** (bool) – True to mark *data* as metadata.

set_log_level(*level: int*) → None

OPTIONAL: Set logging level for the backend and other code.

The default implementation has no effect.

Parameters
level (int) – A Python logging level.

See also:

get_log_level

abstract set_meta(*meta: dict*, *model: str | None*, *scenario: str | None*, *version: int | None*) → None

Set metadata on a target.

Parameters

- **meta** (dict) – Mapping from metadata names/identifiers to values.
- **model** (str or None) – Model name of metadata target.
- **scenario** (str or None) – Scenario name of metadata target.
- **version** (int or None) – *TimeSeries.version* of metadata target.

Raises
ValueError – on unsupported (*model*, *scenario*, *version*) combinations.

See also:

get_meta

abstract set_node(*name: str*, *parent: str | None = None*, *hierarchy: str | None = None*, *synonym: str | None = None*) → None

Add a node name to the Platform.

This method **must** have one of two effects, depending on the arguments:

- With *parent* and *hierarchy*: *name* is added as a child of *parent* in the named *hierarchy*.
- With *synonym*: *synonym* is added as an alias for *name*.

Parameters

- **name** (str) – Node name.
- **parent** (str, *optional*) – Parent node name.
- **hierarchy** (str, *optional*) – Node hierarchy ID.
- **synonym** (str, *optional*) – Synonym for node.

See also:

get_nodes

abstract set_timeslice(*name: str*, *category: str*, *duration: float*) → None

Add a subannual time slice to the Platform.

Parameters

- **name** (str) – Node name.
- **category** (str) – Time slice category.
- **duration** (float) – Time slice duration (a fraction of a year).

See also:

get_timeslices

abstract set_unit(*name: str*, *comment: str*) → None

Add a unit of measurement to the Platform.

Parameters

- **name** (str) – Symbol of the unit.
- **comment** (str) – Description of the change or of the unit.

See also:

get_units

write_file(*path: PathLike*, *item_type:* ItemType, ***kwargs*) → None

OPTIONAL: Write Platform, TimeSeries, or Scenario data to file.

A backend **may** implement write_file for one or more combinations of the *path* and *item_type* methods. For all other combinations, it **must** raise NotImplementedError.

The default implementation supports:

- *path* ending in '.xlsx', *item_type* is either *MODEL* or *SET* | *PAR*: write a single Scenario given by `kwargs['filters']['scenario']` to file using *s_write_excel()*.

Parameters

- **path** (os.PathLike) – File for output. The filename suffix determines the output format.

- **item_type** (*ItemType*) – Type(s) of items to write.

Raises

- **ValueError** – If *ts* is not None and 'scenario' is a key in *filters*.
- **NotImplementedError** – If output of the specified items to the file format is not supported.

See also:

read_file

class ixmp.backend.base.**CachingBackend**(*cache_enabled=True*)

Backend with additional features for caching data.

CachingBackend stores cache values for multiple *TimeSeries*/*Scenario* objects, and for multiple values of a *filters* argument.

Subclasses **must** call *cache()*, *cache_get()*, and *cache_invalidate()* as appropriate to manage the cache; CachingBackend does not enforce any such logic.

_cache: Dict[Tuple, object] = {}

Cache of values. Keys are given by *_cache_key()*; values depend on the subclass' usage of the cache.

_cache_hit: Dict[Tuple, int] = {}

Count of number of times a value was retrieved from cache successfully using *cache_get()*.

classmethod _cache_key(*ts:* TimeSeries, *ix_type: str* | *None*, *name: str* | *None*, *filters: Dict[str, Hashable]* | *None = None*) → Tuple[Hashable, ...]

Return a hashable cache key.

ixmp *filters* (a `dict` of `list`) are converted to a unique id that is hashable.

Returns
A hashable key with 4 elements for *ts*, *ix_type*, *name*, and *filters*.

Return type
`tuple`

cache(*ts:* TimeSeries, *ix_type: str*, *name: str*, *filters: Dict*, *value: Any*) → bool

Store *value* in cache.

Returns
`True` if the key was already in the cache and its value was overwritten.

Return type
`bool`

cache_enabled = True

`True` if caching is enabled.

cache_get(*ts:* TimeSeries, *ix_type: str*, *name: str*, *filters: Dict*) → Any | None

Retrieve value from cache.

The value in *_cache* is copied to avoid cached values being modified by user code. *_cache_hit* is incremented.

Raises
KeyError – If the key for *ts*, *ix_type*, *name* and *filters* is not in the cache.

cache_invalidate(*ts:* TimeSeries, *ix_type: str | None = None*, *name: str | None = None*, *filters: Dict | None = None*) → None

Invalidate cached values.

With all arguments given, single key/value is removed from the cache. Otherwise, multiple keys/values are removed:

- *ts* only: all cached values associated with the *TimeSeries* or *Scenario* object.
- *ts*, *ix_type*, and *name*: all cached values associated with the item, whether filtered or unfiltered.

del_ts(*ts:* TimeSeries)

Invalidate cache entries associated with *ts*.

Backend API.

ixmp.backend.**FIELDS = {'get_nodes': ('region', 'mapped_to', 'parent', 'hierarchy'), 'get_scenarios': ('model', 'scenario', 'scheme', 'is_default', 'is_locked', 'cre_user', 'cre_date', 'upd_user', 'upd_date', 'lock_user', 'lock_date', 'annotation', 'version'), 'get_timeslices': ('name', 'category', 'duration'), 'ts_get': ('region', 'variable', 'unit', 'subannual', 'year', 'value'), 'ts_get_geo': ('region', 'variable', 'subannual', 'year', 'value', 'unit', 'meta'), 'write_file': ('MODEL', 'SCENARIO', 'VERSION', 'VARIABLE', 'UNIT', 'REGION', 'META', 'SUBANNUAL', 'YEAR', 'VALUE')}**

Lists of field names for tuples returned by Backend API methods.

The key "write_file" refers to the columns appearing in the CSV output from *export_timeseries_data()* when using *JDBCBackend*.

Todo: Make this consistent with other dimension orders and with *IAMC_IDX*.

---

ixmp.backend.**IAMC_IDX: List[str | int] = ['model', 'scenario', 'region', 'variable', 'unit']**

Partial list of dimensions for the IAMC data structure, or "IAMC format". This omits "year" and "subannual" which appear in some variants of the structure, but not in others.

class ixmp.backend.**ItemType**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Type of data items in *TimeSeries* and *Scenario*.

TS = 1

T = 1

Time series data variable.

SET = 2

S = 2

Set.

PAR = 4

P = 4

Parameter.

VAR = 8

V = 8
Model variable.

EQU = 16

E = 16
Equation.

MODEL = 30

M = 30
All kinds of model-related data, i.e. *SET*, *PAR*, *VAR* and *EQU*.

SOLUTION = 24
Model solution data, i.e. *VAR* and *EQU*.

ALL = 31

A = 31
All data, i.e. *MODEL* and *TS*.

2.3.3 Common input/output routines for backends

ixmp.backend.io.**EXCEL_MAX_ROWS = 1048576**
Maximum number of rows supported by the Excel file format. See *to_excel()* and *Scenario/model data*.

ixmp.backend.io.**maybe_init_item**(*scenario*, *ix_type*, *name*, *new_idx*, *path*)
Call *init_set()*, *init_par()*, etc. if possible.

Logs an intelligible warning and then raises ValueError in two cases:

- the *new_idx* is ambiguous, e.g. containing index names that cannot be used to infer index sets, or
- an existing item has index names that are different from *new_idx*.

ixmp.backend.io.**s_read_excel**(*be*, *s*, *path*, *add_units=False*, *init_items=False*, *commit_steps=False*)
Read data from a Microsoft Excel file at *path* into *s*.

See also:

Scenario.read_excel

ixmp.backend.io.**s_write_excel**(*be*, *s*, *path*, *item_type*, *filters=None*, *max_row=None*)
Write *s* to a Microsoft Excel file at *path*.

See also:

Scenario.to_excel

ixmp.backend.io.**ts_read_file**(*ts*, *path*, *firstyear=None*, *lastyear=None*)
Read data from a CSV or Microsoft Excel file at *path* into *ts*.

See also:

TimeSeries.add_timeseries, *TimeSeries.read_file*

2.4 File formats and input/output

In addition to the data management features provided by *Storage back ends (ixmp.backend)*, ixmp is able to write and read `TimeSeries` and `Scenario` data to and from files. This page describes those options and formats.

2.4.1 Time series data

Time series data can be:

- Read using `TimeSeries.read_file()`, or the *CLI command* `ixmp import timeseries FILE` for a single TimeSeries object.
- Written using `export_timeseries_data()` for multiple TimeSeries objects at once.

Both CSV and Excel files in the IAMC time-series format are supported.

2.4.2 Scenario/model data

Scenario data can be read from/written to Microsoft Excel files using `Scenario.read_excel()` and `to_excel()`, and the CLI commands `ixmp import scenario FILE` and `ixmp export FILE`. The files have the following structure:

- One sheet named 'ix_type_mapping' with two columns:
  - 'item': the name of an ixmp item.
  - 'ix_type': the item's type as a length-3 string: 'set', 'par', 'var', or 'equ'.
- One or more sheet per item. If the length of data is greater than the maximum number of rows per sheet supported by the Excel file format (`EXCEL_MAX_ROWS`), the item is split across multiple sheets named, e.g., 'foo', 'foo(2)', 'foo(3)'.
- Sets:
  - Sheets for one-dimensional indexed sets have one column, with a header cell that is the index set name.
  - Sheets for multi-dimensional indexed sets have multiple columns.
  - Sets with no elements are represented by empty sheets.
- Parameters, variables, and equations:
  - Sheets have zero (for scalar items) or more columns with headers that are the index *names* (not necessarily sets; see below) for those dimensions.
  - Parameter sheets have 'value' and 'unit' columns.
  - Variable and equation sheets have 'lvl' and 'mrg' columns.
  - Items with no elements are not included in the file.

Limitations

Reading variables and equations
The ixmp API provides no way to set the data of variables and equations, because these are considered model solution data.

Thus, while *to_excel()* will write files containing variable and equation data, *read_excel()* can not add these to a Scenario, and only emits log messages indicating that they are ignored.

Multiple dimensions indexed by the same set
read_excel() provides the *init_items* argument to create new sets and parameters when reading a file. However, the file format does not capture information needed to reconstruct the original data in all cases.

For example:

```
scenario.init_set('foo')
scenario.add_set('foo', ['a', 'b', 'c'])
scenario.init_par(name='bar', idx_sets=['foo'])
scenario.init_par(
    name='baz',
    idx_sets=['foo', 'foo'],
    idx_names=['foo', 'another_dimension'])
scenario.to_excel('file.xlsx')
```

`file.xlsx` will contain sheets named 'bar' and 'baz'. The sheet 'bar' will have column headers 'foo', 'value', and 'unit', which are adequate to reconstruct the parameter. However, the sheet 'baz' will have column headers 'foo' and 'another_dimension'; this information does not allow ixmp to infer that 'another_dimension' is indexed by 'foo'.

To work around this limitation, initialize 'baz' with the correct dimensions before reading its data:

```
new_scenario.init_par(
    name='baz',
    idx_sets=['foo', 'foo'],
    idx_names=['foo', 'another_dimension'])
new_scenario.read_excel('file.xlsx', init_items=True)
```

File formats other than `.xlsx`
The `.xlsx` (Office Open XML) file format is preferred for input and output. *ixmp* uses openpyxl and `pandas` in order to read and write this format. For other Excel file formats, including `.xls` and `.xlsb`, see the Pandas documentation.

2.5 Mathematical models (`ixmp.model`)

By default, the *ix modeling platform* is installed with *ixmp.model.gams.GAMSModel*, which performs calculations by executing code stored in GAMS files.

However, *ix modeling platform* is extensible to support other methods of performing calculations or optimization. Developers wishing to add such capabilities may subclass *ixmp.model.base.Model* and implement its methods.

2.5.1 Provided models

ixmp.model.**MODELS**: Dict[str, Type] = {'dantzig': <class 'ixmp.model.dantzig.DantzigModel'>, 'default': <class 'ixmp.model.gams.GAMSModel'>, 'gams': <class 'ixmp.model.gams.GAMSModel'>}

Mapping from names to available models. To register additional models, add elements to this variable.

ixmp.model.**get_model**(*name*, ***model_options*)

Return a model for *name* (or the default) with *model_options*.

class ixmp.model.gams.**GAMSModel**(*name_=None*, ***model_options*)

General class for ixmp models using GAMS.

GAMSModel solves a Scenario using the following steps:

1. All Scenario data is written to a model input file in GDX format.
2. A GAMS program is run to perform calculations, producing output in a GDX file.
3. Output, or solution, data is read from the GDX file and stored in the Scenario.

When created and *run()*, GAMSModel constructs file paths and other necessary values using format strings. The *defaults* may be overridden by the keyword arguments to the constructor:

Parameters

- **name** (str, *optional*) – Override the *name* attribute to provide the *model_name* for format strings.
- **model_file** (str, *optional*) – Path to GAMS file, including '.gms' extension. Default: '{model_name}.gms' in the current directory.
- **case** (str, *optional*) – Run or case identifier to use in GDX file names. Default: '{scenario.model}_{scenario.name}', where *scenario* is the *Scenario* object passed to *run()*. Formatted using *model_name* and *scenario*.
- **in_file** (str, *optional*) – Path to write GDX input file. Default: '{model_name}_in.gdx'. Formatted using *model_name*, *scenario*, and *case*.
- **out_file** (str, *optional*) – Path to read GDX output file. Default: '{model_name}_out.gdx'. Formatted using *model_name*, *scenario*, and *case*.
- **solve_args** (list of str, *optional*) – Arguments to be passed to GAMS, e.g. to identify the model input and output files. Each formatted using *model_file*, *scenario*, *case*, *in_file*, and *out_file*. Default:
  - '--in="{in_file}"'
  - '--out="{out_file}"'
- **gams_args** (list of str, *optional*) – Additional arguments passed directly to GAMS without formatting, e.g. to control solver options or behaviour. See the GAMS Documentation. For example:
  - ["iterLim=10"] limits the solver to 10 iterations.
- **quiet** (bool, *optional*) – If True, add "LogOption=2" to *gams_args* to redirect most console output during the model run to the log file. Default False, so "LogOption=4" is added. Any "LogOption" value provided explicitly via *gams_args* takes precedence.
- **check_solution** (bool, *optional*) – If True, raise an exception if the GAMS solver did not reach optimality. (Only for MESSAGE-scheme Scenarios.)

- **comment** (str, *optional*) – Comment added to Scenario when importing the solution. If omitted, no comment is added.
- **equ_list** (list of str, *optional*) – Equations to be imported from the *out_file*. Default: all.
- **var_list** (list of str, *optional*) – Variables to be imported from the *out_file*. Default: all.

defaults: MutableMapping[str, Any] = {'case': '{scenario.model}_{scenario.scenario}', 'check_solution': True, 'comment': None, 'equ_list': None, 'gams_args': [], 'in_file': '{cwd}/{model_name}_in.gdx', 'model_file': '{model_name}.gms', 'out_file': '{cwd}/{model_name}_out.gdx', 'quiet': False, 'solve_args': ['--in="{in_file}"', '--out="{out_file}"'], 'use_temp_dir': True, 'var_list': None}

Default values and format strings for options.

format(*value*)

Helper for recursive formatting of model options.

value is formatted with replacements from the attributes of *self*.

format_exception(*exc*, *model_file*, *backend_class*)

Format a user-friendly exception when GAMS errors.

format_option(*name*)

Retrieve the option *name* and format it.

name: str = 'default'

Model name.

remove_temp_dir(*msg='after run()'*)

Remove the temporary directory, if any.

run(*scenario*)

Execute the model.

ixmp.model.gams.**RETURN_CODE = {0: 'Normal return', 1: 'Solver is to be called, the system should never return this number', 2: 'There was a compilation error', 3: 'There was an execution error', 4: 'System limits were reached', 5: 'There was a file error', 6: 'There was a parameter error', 7: 'There was a licensing error', 8: 'There was a GAMS system error', 9: 'GAMS could not be started', 10: 'Out of memory', 11: 'Out of disk', 109: 'Could not create process/scratch directory', 110: 'Too many process/scratch directories', 112: 'Could not delete the process/scratch directory', 113: 'Could not write the script gamsnext', 114: 'Could not write the parameter file', 115: 'Could not read environment variable', 136: 'Driver error: internal error: cannot load option handling library', 141: 'Cannot add path / unknown UNIX environment / cannot set environment variable', 144: 'Could not spawn the GAMS language compiler (gamscmex)', 145: 'Current directory (curdir) does not exist', 146: 'Cannot set current directory (curdir)', 148: 'Blank in system directory', 149: 'Blank in current directory', 150: 'Blank in scratch extension (screxT)', 151: 'Unexpected cmexRC', 152: 'Could not find the process directory (procdir)', 153: 'CMEX library not be found (experimental)', 154: 'Entry point in CMEX library could not be found (experimental)', 155: 'Blank in process directory', 156: 'Blank in scratch directory', 160: 'Driver error: internal error: GAMS compile and execute module not found', 184: 'Driver error: problems getting current directory', 208: 'Driver error: internal error: cannot install interrupt handler', 232: 'Driver error: incorrect command line parameters for gams'}**

Return codes used by GAMS, from https://www.gams.com/latest/docs/UG_GAMSReturnCodes.html . Values over 256 are only valid on Windows, and are returned modulo 256 on other platforms.

ixmp.model.gams.**gams_version**()

Return the GAMS version as a string, e.g. '24.7.4'.

class ixmp.model.dantzig.**DantzigModel**(*name_=None*, ***model_options*)

Dantzig's cannery/transport problem as a *GAMSModel*.

Provided for testing *ixmp* code.

defaults: MutableMapping[str, Any] = ChainMap({'model_file': PosixPath('/home/docs/checkouts/readthedocs.org/user_builds/iiasa-energy-program-ixmp/envs/latest/lib/python3.11/site-packages/ixmp/model/dantzig.gms')}, {'model_file': '{model_name}.gms', 'case': '{scenario.model}_{scenario.scenario}', 'in_file': '{cwd}/{model_name}_in.gdx', 'out_file': '{cwd}/{model_name}_out.gdx', 'solve_args': ['--in="{in_file}"', '--out="{out_file}"'], 'gams_args': [], 'check_solution': True, 'comment': None, 'equ_list': None, 'var_list': None, 'quiet': False, 'use_temp_dir': True})

Default values and format strings for options.

classmethod initialize(*scenario*, *with_data=False*)

Initialize the problem.

If *with_data* is `True` (default: `False`), the set and parameter values from the original problem are also populated. Otherwise, the sets and parameters are left empty.

name: str = 'dantzig'

Model name.

2.5.2 Model API

exception ixmp.model.base.**ModelError**

Error in model code—that is, *Model.run()* or other code called by it.

class ixmp.model.base.**Model**(*name*, ***kwargs*)

In the following, the words **required**, **optional**, etc. have specific meanings as described in IETF RFC 2119.

Model is an *abstract* class; this means it **must** be subclassed. It has two **required** methods that **must** be overridden by subclasses:

__init__(name, **kwargs)	Constructor.
run(scenario)	Execute the model.

The following attributes and methods are **optional** in subclasses. The default implementations are either empty or implement reasonable default behaviour.

enforce(scenario)	Enforce data consistency in *scenario*.
initialize(scenario)	Set up *scenario* with required items.
initialize_items(scenario, items)	Helper for *initialize()*.
name	Name of the model.

abstract __init__(*name*, ***kwargs*)

Constructor.

Required.

Parameters
kwargs – Model options, passed directly from *Scenario.solve()*.

Model subclasses MUST document acceptable option values.

static enforce(*scenario*)

Enforce data consistency in *scenario*.

Optional; the default implementation does nothing. Subclass implementations of *enforce()*:

- **should** modify the contents of sets and parameters so that *scenario* contains structure and data that is consistent with the underlying model.
- **must not** add or remove sets or parameters; for that, use *initialize()*.

enforce() is always called by *run()* before the model is run or solved; it **may** be called manually at other times.

Parameters
scenario (*Scenario*) – Object on which to enforce data consistency.

classmethod initialize(*scenario*)

Set up *scenario* with required items.

Optional; the default implementation does nothing. Subclass implementations of *initialize()*:

- **may** add sets, set elements, and/or parameter values.
- **may** accept any number of keyword arguments to control behaviour.
- **must not** modify existing parameter data in *scenario*, either by deleting or overwriting values; for that, use *enforce()*.

Parameters
scenario (*Scenario*) – Object to initialize.

See also:

initialize_items

classmethod initialize_items(*scenario:* Scenario, *items: Mapping[str, Dict]*) → None

Helper for *initialize()*.

All of the *items* are added to *scenario*. Existing items are not modified. Errors are logged if the description in *items* conflicts with the index set(s) and/or index name(s) of existing items.

initialize_items may perform one commit. *scenario* is in the same state (checked in, or checked out) after initialize_items is complete.

Parameters

- **scenario** (*Scenario*) – Object to initialize.
- **items** – Keys are names of ixmp items (set, parameter, equation, or variable) to initialize. Values are `dict`, and each **must** have the key 'ix_type' (one of 'set', 'par', 'equ', or 'var'); any other entries are keyword arguments to the corresponding methods such as *init_set()*.

Raises
ValueError – if *scenario* has a solution, i.e. *has_solution()* is True.

See also:

init_equ, *init_par*, *init_set*, *init_var*

name: str = 'base'
Name of the model.

abstract run(*scenario*)
Execute the model.

Required. Implementations of *run()*:

- **must** call *enforce()*.

Parameters
scenario (*Scenario*) – Scenario object to solve by running the Model.

2.6 Reporting / postprocessing

ixmp.report provides features for computing derived values from the contents of a *ixmp.Scenario*, *after* it has been solved using a model and the solution data has been stored. It is built on the genno package, which has its own, separate documentation. This page provides only API documentation.

- For an introduction and basic concepts, see Concepts and usage in the genno documentation.
- For automatic reporting of message_ix.Scenario, see Postprocessing and reporting in the MESSAGE*ix* documentation.

2.6.1 Top-level classes and functions

The following top-level objects from genno may also be imported from *ixmp.report*.

ComputationError(exc)	Wrapper to print intelligible exception information for Computer.get().
Key(name_or_value[, dims, tag, _fast])	A hashable key for a quantity that includes its dimensionality.
KeyExistsError	Raised by Computer.add() when the target key exists.
MissingKeyError	Raised by Computer.add() when a required input key is missing.
Quantity(*args, **kwargs)	A sparse data structure that behaves like xarray.DataArray.
configure([path])	Configure genno globally.

ixmp.report additionally defines:

Reporter(*args, **kwargs)	Class for describing and executing computations.

class ixmp.report.**Reporter**(**args*, ***kwargs*)

Class for describing and executing computations.

A Reporter extends a genno.Computer to postprocess data from one or more *ixmp.Scenario* objects.

Using the *from_scenario()*, a Reporter is automatically populated with:

- Keys that retrieve the data for every *ixmp* item (parameter, variable, equation, or scalar) available in the Scenario.

finalize(scenario)	Prepare the Reporter to act on *scenario*.
from_scenario(scenario, **kwargs)	Create a Reporter by introspecting *scenario*.
set_filters(**filters)	Apply *filters* ex ante (before computations occur).

The Computer class provides the following methods:

add(data, *args, **kwargs)	General-purpose method to add computations.
add_queue(queue[, max_tries, fail])	Add tasks from a list or *queue*.
add_single(key, *computation[, strict, index])	Add a single *computation* at *key*.
apply(generator, *keys, **kwargs)	Add computations by applying *generator* to *keys*.
check_keys(*keys[, predicate, action])	Check that *keys* are in the Computer.
configure([path, fail, config])	Configure the Computer.
describe([key, quiet])	Return a string describing the computations that produce *key*.
full_key(name_or_key)	Return the full-dimensionality key for *name_or_key*.
get([key])	Execute and return the result of the computation *key*.
infer_keys(key_or_keys[, dims])	Infer complete *key_or_keys*.
keys()	Return the keys of `graph`.
visualize(filename[, key, optimize_graph])	Generate an image describing the Computer structure.
write(key, path)	Compute *key* and write the result directly to *path*.

The following methods are deprecated; equivalent or better functionality is available through `Computer.add()`. See the genno documentation for each method for suggested changes/migrations.

add_file(*args, **kwargs)	Deprecated.
add_product(*args, **kwargs)	Deprecated.
aggregate(qty, tag, dims_or_groups[, ...])	Deprecated.
convert_pyam(*args, **kwargs)	Deprecated.
disaggregate(qty, new_dim[, method, args])	Deprecated.

finalize(*scenario:* Scenario) → None

Prepare the Reporter to act on *scenario*.

The *TimeSeries* (thus also *Scenario* or message_ix.Scenario) object *scenario* is stored with the key `'scenario'`. All subsequent processing will act on data from this Scenario.

classmethod from_scenario(*scenario:* Scenario, ***kwargs*) → *Reporter*

Create a Reporter by introspecting *scenario*.

Parameters

- **scenario** (*Scenario*) – Scenario to introspect in creating the Reporter.
- **kwargs** – Passed to `genno.Computer.configure()`.

Returns

A Reporter instance containing:

- A 'scenario' key referring to the *scenario* object.
- Each parameter, equation, and variable in the *scenario*.
- All possible aggregations across different sets of dimensions.
- Each set in the *scenario*.

Return type
Reporter

set_filters(***filters*) → None

Apply *filters* ex ante (before computations occur).

See the description of *filters()* under *Configuration*.

2.6.2 Configuration

ixmp.report adds handlers for two configuration sections, and modifies the behaviour of one from genno

ixmp.report.**filters**(*c: Computer*, *filters: dict*)

Handle the entire `filters:` config section.

Reporter-specific configuration.

Affects data loaded from a Scenario using *data_for_quantity()*, which filters the data before any other computation takes place. Filters are stored at `Reporter.graph["config"]["filters"]`.

If no arguments are provided, *all* filters are cleared. Otherwise, *filters* is a mapping of `str` → (`list` of `str` or `None`. Keys are dimension IDs. Values are either lists of allowable labels along the respective dimension or `None` to clear any existing filters for that dimension.

This configuration can be applied through *Reporter.set_filters()*; `Reporter.configure`, or in a configuration file:

```
filters:
  # Exclude a label "x2" on the "x" dimension, etc.
  x: [x1, x3, x4]
  technology: [coal_ppl, wind_ppl]
  # Clear existing filters for the "commodity" dimension
  commodity: null
```

report.**rename_dims**(*info: dict*)

Handle the entire `rename_dims:` config section.

Reporter-specific configuration.

Affects data loaded from a Scenario using *data_for_quantity()*. Native dimension names are mapped; in the example below, the dimension "i" is present in the Reporter as "i_renamed" on all quantities/keys in which it appears.

```
rename_dims:
  i: i_renamed
```

report.**units**(*info: dict*)

Handle the entire `units:` config section.

The only difference from genno.config.units() is that this handler keeps the configuration values stored in `Reporter.graph["config"]`. This is so that *data_for_quantity()* can make use of `["units"]["apply"]`

2.6.3 Operators

ixmp.report defines these operators:

data_for_quantity(ix_type, name, column, ...)	Retrieve data from *scenario*.
from_url(url[, cls])	Return a *TimeSeries* or subclass instance, given its *url*.
get_ts(ts[, filters, iamc, subannual])	Retrieve timeseries data from *ts*.
map_as_qty(set_df, full_set)	Convert *set_df* to a Quantity.
update_scenario(scenario, *quantities[, params])	Update *scenario* with computed data from reporting *quantities*.
store_ts(scenario, *data[, strict])	Store time series *data* on *scenario*.
remove_ts(ts[, data, after])	Remove all time series data from *ts*.

Basic operators are defined by genno.operator and its compatibility modules; see there for details:

Plot()	Class for plotting using `plotnine`.
add	Sum across multiple *quantities*.
aggregate(quantity, groups, keep)	Aggregate *quantity* by *groups*.
apply_units(qty, units)	Apply *units* to *qty*.
as_pyam	Return a `pyam.IamDataFrame` containing the data from *quantity*.
broadcast_map(quantity, map[, rename, strict])	Broadcast *quantity* using a *map*.
combine(*quantities[, select, weights])	Sum distinct *quantities* by *weights*.
concat(*objs, **kwargs)	Concatenate Quantity *objs*.
disaggregate_shares(quantity, shares)	Deprecated: Disaggregate *quantity* by *shares*.
div	Compute the ratio *numerator* / *denominator*.
group_sum(qty, group, sum)	Group by dimension *group*, then sum across dimension *sum*.
interpolate(qty[, coords, method, ...])	Interpolate *qty*.
load_file	Read the file at *path* and return its contents as a `Quantity`.
mul	Compute the product of any number of *quantities*.
pow(a, b)	Compute *a* raised to the power of *b*.
product	Alias of `mul()`, for backwards compatibility.
relabel(qty[, labels])	Replace specific labels along dimensions of *qty*.
rename_dims(qty[, new_name_or_name_dict])	Rename the dimensions of *qty*.
ratio	Alias of `div()`, for backwards compatibility.
select(qty, indexers, *[, inverse, drop])	Select from *qty* based on *indexers*.
sum	Sum *quantity* over *dimensions*, with optional *weights*.
write_report(quantity, path)	Write a quantity to a file.

ixmp.report.operator.**data_for_quantity**(*ix_type: Literal['equ', 'par', 'var']*, *name: str*, *column: Literal['mrg', 'lvl', 'value']*, *scenario:* Scenario, *config: Mapping[str, Mapping]*) → Quantity

Retrieve data from *scenario*.

Parameters

- **ix_type** – Type of the ixmp object.
- **name** (str) – Name of the ixmp object.
- **column** – Data to retrieve. 'mrg' and 'lvl' are valid only for `ix_type='equ'`,and 'level' otherwise.
- **scenario** (*ixmp.Scenario*) – Scenario containing data to be retrieved.
- **config** – Configuration. The key 'filters' may contain a mapping from dimensions to iterables of allowed values along each dimension. The key 'units'/'apply' may contain units to apply to the quantity; any such units overwrite existing units, without conversion.

Returns
Data for *name*.

Return type
Quantity

ixmp.report.operator.**from_url**(*url: str*, *cls=<class 'ixmp.core.timeseries.TimeSeries'>*) → *TimeSeries*

Return a *TimeSeries* or subclass instance, given its *url*.

Parameters
cls (type, *optional*) – Subclass to instantiate and return; for instance, *Scenario*.

ixmp.report.operator.**get_ts**(*ts:* TimeSeries, *filters: dict | None = None*, *iamc: bool = False*, *subannual: bool | str = 'auto'*) → DataFrame

Retrieve timeseries data from *ts*.

Corresponds to *TimeSeries.timeseries()*.

Parameters

filters – Names and values for the *region*, *variable*, *unit*, and *year* keyword arguments to *timeseries()*.

ixmp.report.operator.**map_as_qty**(*set_df: DataFrame*, *full_set*)

Convert *set_df* to a Quantity.

For the MESSAGE sets named `cat_*` (see Category types and mappings) *ixmp.Scenario.set()* returns a DataFrame with two columns: the *category* set (S1) elements and the *category member* set (S2, also required as the argument *full_set*) elements.

map_as_qty converts such a DataFrame (*set_df*) into a Quantity with two dimensions. At the coordinates *(s_1, s_2)*, the value is 1 if s_2 is mapped from s_1; otherwise 0.

A category named 'all', containing all elements of *full_set*, is added automatically.

See also:

broadcast_map

ixmp.report.operator.**remove_ts**(*ts:* TimeSeries, *data: DataFrame | None = None*, *after: int | None = None*) → None

Remove all time series data from *ts*.

Note that data stored with *add_timeseries()* using `meta=True` as a keyword argument cannot be removed using *TimeSeries.remove_timeseries()*, and thus also not with this operator.

Parameters

- **data** (pandas.DataFrame, *optional*) – Specific data to be removed. If not given, all time series data is removed.
- **after** (int, *optional*) – If given, only data with *year* labels equal to or greater than *after* are removed.

ixmp.report.operator.**store_ts**(*scenario*, **data*, *strict: bool = False*) → None

Store time series *data* on *scenario*.

The data is stored using *add_timeseries()*; *scenario* is checked out and then committed.

Parameters

- **scenario** – Scenario on which to store data.
- **data** (pandas.DataFrame or pyam.IamDataFrame) – 1 or more objects containing data to store. If pandas.DataFrame, the data are passed through *to_iamc_layout()*.
- **strict** (bool) – If True (default False), raise an exception if any of *data* are not successfully added. Otherwise, log on level ERROR and continue.

ixmp.report.operator.**update_scenario**(*scenario*, **quantities*, *params=[]*)

Update *scenario* with computed data from reporting *quantities*.

Parameters

- **scenario** (*Scenario*) –

- **quantities** (genno.Quantity or pandas.DataFrame) – If DataFrame, must be valid input to *Scenario.add_par()*.
- **params** (list of str, *optional*) – For every element of *quantities* that is a pd.DataFrame, the element of *params* at the same index gives the name of the parameter to update.

2.6.4 Utilities

ixmp.report.common.**RENAME_DIMS: Dict[str, str] = {}**

Dimensions to rename when extracting raw data from Scenario objects. Mapping from Scenario dimension name -> preferred dimension name.

User code **should** avoid directly manipulating *RENAME_DIMS*. Instead, call `configure()`:

```
# Rename dimension "long_dimension_name" to "ldn"
configure(rename_dims={"long_dimension_name": "ldn"})
```

As well, importing the variable into the global namespace of another module creates a copy of the dictionary that may become out of sync with other changes. Thus, instead of:

```
from ixmp.report import RENAME_DIMS

def my_operator(...):
    # Code that references RENAME_DIMS
```

Do this:

```
def my_operator(...):
    from ixmp.report import common

    # Code that references common.RENAME_DIMS
```

ixmp.report.util.**dims_for_qty**(*data*)

Return the list of dimensions for *data*.

If *data* is a pandas.DataFrame, its columns are processed; otherwise it must be a list.

RENAME_DIMS is used to rename dimensions.

ixmp.report.util.**keys_for_quantity**(*ix_type*, *name*, *scenario*)

Return keys for *name* in *scenario*.

2.7 Data model

ixmp stores many types of data objects. This page describes the ixmp *data model*,[1] with each kind of object in its own section. The description here is **application independent**: it describes how *ixmp* handles data without reference to specific uses of *ixmp* to model specific real-world systems (such as `message_ix`).

Note: Due to the development history of *ixmp*, some words are used with 2 or more different meanings in the data model, including 'model', 'scenario', 'time series', 'meta', 'category', and 'region/node'. This page lists and

[1] In this sense, the word "model" refers to how data is structured, and its meaning, within an *ixmp.Platform*; this is distinct from a specific numeric/optimization model that can be used to solve an *ixmp.Scenario* (for that, see *Mathematical models (ixmp.model)*).

disambiguates the multiple meanings.

2.7.1 Top-level classes

Platform

- A Platform is identified by a string **name**.
  - This name is a reference to configuration stored in the ixmp *configuration file*. Because this file is local to each user's system, the same platform name on different systems may refer to different stored data or different local or remote databases.
- A Platform contains:
  - Zero or more TimeSeries and Scenario objects.
  - *Other data associated with the Platform*, but not specific to any TimeSeries or Scenario.
- A Platform stores data using a back end. This data may be in a file, a **database** (local or remote), in memory, or elsewhere.

TimeSeries (object)

- Each TimeSeries is uniquely identified by three attributes:
  - **model name**: an arbitrary string. This may refer to a model, modeling team, individual, or other data source that produced the data stored in the TimeSeries—even if that model is not implemented in *ixmp*.
  - **scenario name**: an arbitrary string. For the same "model name", the scenario name can be used to distinguish multiple alternate scenarios, narratives, counterfactuals, cases, etc. created using different settings or input data, using the same model or from the same data source.
  - **version**: an integer.
- TimeSeries also have a "run ID". This is a value in 1:1 correspondence with the unique (model name, scenario name, version) identifiers.
- There is no guarantee that a given (model name, scenario name, version) on one Platform refers to a TimeSeries with the same data as the same identifiers on another Platform.
- There is no correspondence between any two TimeSeries with the same model name and scenario name, but different versions. The two may contain entirely different data.
- For each combination of (model name, scenario name), one version may be set set as the **default version**.

Scenario (object)

- Scenario is a subclass of TimeSeries and inherits its behaviour. This means that:
  - All of the above statements about TimeSeries objects also apply to Scenario objects.
  - All kinds of *data associated with a TimeSeries object* can also be stored within a Scenario object.
  - All statements below about TimeSeries objects also apply to Scenario objects.
- Scenarios additionally have a:
  - **scheme**: a string, that may (but does not necessarily) refer to a particular mathematical model used to solve or run the scenario, and/or corresponding list of *items* to which the Scenario data conforms.

    For example: the scheme "MESSAGE" refers to `message_ix.models.MESSAGE`, its mathematical model, and particular items.

2.7.2 Data associated with a Platform

Lists of identifiers

- A Platform stores 7 specific lists of identifiers.
- Each identifier is a string.
- Some lists have additional attributes associated with each identifier.
- Some values are pre-populated, i.e. always present on a new Platform.
- For 5 of these lists, identifiers can be added, removed, and are referenced in various ways by data in other kinds of objects.
- In some cases, they are automatically populated based on other data manipulations.

Model name
Values for the "model name" identifier of TimeSeries objects on the Platform, as described above.

This list is automatically extended with any model name used for a new TimeSeries, but may also contain values that are not used by any existing TimeSeries object.

Scenario name
Values for the "scenario name" identifier of TimeSeries objects on the Platform, as described above.

This list is automatically extended with any scenario name used for a new TimeSeries, but may also contain values that are not used by any existing TimeSeries object.

Unit
Units of measurement.

Values for:

- the "unit" identifier of time-series and geodata in TimeSeries objects on the Platform.
- the "unit" attribute of parameter data in Scenario objects on the Platform.

Region
A geographic region or area, e.g. country, multi-country- or sub-national region, city, etc.

Values for the "region" identifier of time-series and geodata in TimeSeries objects on the Platform.

In addition to its ID string, each identifier has the following attributes:

- **hierarchy**: a string, identifying 1 of multiple possible sets of of parent/child relationships.
- **parent**: a string, optional, giving the identifier of a region which is the parent of the region.
- **mapped_to**: a string, optional, giving the identifier of another region for which the identifier is an alias.

Sub-annual time slice
Portion of a calendar year.[3]

Values for the "subannual" identifier of time-series and geodata in TimeSeries objects on the Platform.

In addition to its ID string, each identifier has the following attributes:

- **duration**: a float number indicating the duration of the time slice, expressed in fraction of a year (dimensionless).
- **category**: a string, identifying a set of time slices that together represent a division of one year.

The value "Year" is automatically present, with duration `1.0`. Use of this value for the "subannual" identifier indicates that the time-series or geodata **does not** have subannual resolution.

(Metadata)
These are the name or ID of metadata entries; see *Metadata*, below.

This list is not directly modifiable.

(Variable)
These are values that may appear for the "variable" identifier of time-series or geodata in TimeSeries objects on the Platform.

This list is not directly modifiable.

[3] The concept of a time slice is related to the concept represented by the index set 'time' in a `message_ix.Scenario` to indicate a subannual time dimension. However, these are not linked automatically within *`ixmp`* or `message_ix` and must be defined independently. See Years, periods, and time slices.

Metadata

- These are a key-value store for arbitrary metadata.
- Each entry is uniquely identified by:
  - a **"meta name"** or **ID**: an arbitrary string.
- In addition each entry has:
  - a **value**: either a string, a number (floating-point, integer, or boolean), or a list of these.
  - the **target** to which it is attached or associated. This may be one of:
    1. A set of (model name, scenario name, version).
    2. A set of (model name, scenario name).
    3. A model name.
    4. A scenario name.
- As an artifact of some early applications, terms including "category" and "(quantitative) indicator" are variously used for the metadata identifier or metadata value. The term "level" is sometimes used to refer to the different kinds of targets.
- Because the name is the unique identifier, the same name cannot be used with different targets.
- The model name and/or scenario name to which an entry is associated **must** be in the *Lists of identifiers* on the Platform. It is not required that any specific TimeSeries exist that are identified by these model name(s) and/or scenario name(s).

Documentation

- This is a second kind of key-value store for arbitrary metadata.
- Each entry is uniquely identified by:
  - A **domain**: one of "scenario", "model", "region", "metadata", "timeseries".
  - An **identifier**. Depending on the domain, this must be a value from one of the *lists of identifiers*:

Domain	Identifier appears in the list…
model	Model name
scenario	Scenario name
region	Region
metadata	Meta, i.e. the name/ID used by *metadata* entries
timseries	Variable, i.e. values for the "variable" identifier of time-series or geodata

- Each entry consists of a string, e.g. containing a block of text.

2.7.3 Data associated with a TimeSeries object

Time series data

- A TimeSeries object may contain zero or more series of time-series data.
- "series" means a 1-dimensional vector of numerical data.
- "time" means that the single dimension, called **year**, refers to a time period: either a calendar year or an identifying year in a multi-year period.

  Thus, the series consists of a mapping from years to numerical values.
- Each series is identified by:
  - **variable** name: an abitrary string.
  - **region**: a value from the "Region" list (see *Lists of identifiers*).
  - **unit**: a value from the "Unit" list (see *Lists of identifiers*).
  - **subannual**: a value from the "Sub-annual time slices" list (see *Lists of identifiers*).
  - **meta**: a boolean value.

    Note: This is distinct from *Metadata*, above.

Geodata

- These are identical to time-series data, except the individual values are strings instead of numbers.
- The content and meaning of the strings are user-determined.
- The name "geodata" is an artifact of the initial use-case: to store URIs or other references to geographic information systems (GIS) data, stored separately from the Platform.

2.7.4 Data associated with a Scenario object

Item

- A Scenario object may contain zero or more **items**.
- Each item is uniquely identified by a string **name**.
- Each item addtionally has a **type** (`ItemType`):
  - This is one of set (`SET`), parameter (`PAR`), variable (`VAR`), or equation (`EQU`).
  - This distinction is based on the data model common in algebraic modeling languages such as GAMS, Pyomo, and others.
- Because the name is the unique identifier, item names are unique *across all item types* within the same Scenario. For instance, it is not possible to have both a 'set' item and 'parameter' item with the name "foo".
- The term **model data** (`MODEL`) refers to any type of item.
- The term **model solution data** (`SOLUTION`) refers to variable or equation data.

  Because these items are populated with data when a model is solved or run, a Scenario that contains any values for any item with either of these types is said to "contain a model solution".

Set

- A *set* is either:
  1. a simple set, or
  2. an indexed set.[2]
- A **simple**, **basic**, or **index set** is a list of strings.
- An **indexed set** has:
  - 1 or more **dimensions**.

    Each dimension is associated with a simple set, and has an optional string **name**. Specific values for each dimension/index set of an indexed set comprise a **key**.
  - boolean **values**.

    Each element of a index set (or each key, comprising values for 2 or more index sets) either is, or is not, a member of the indexed set.

Pararameter

- Parameters, variables, and equations is **indexed** by 0 or more simple sets, and thus have index sets and dimension names in the same way as described above for indexed sets.
- A parameter, variable, or equation indexed by 0 sets (0-dimensional) is a **scalar**.
- For each each key, a parameter has:
  - A single numeric **value**.
  - A **unit** attribute.

    The values of this attribute must be in the *"Unit" list* of the Platform containing the Scenario containing the parameter.

Variable, equation

- Variables and equations have two numeric values for each key:
  - **level**: the actual value of the variable/equation.
  - **marginal**: the change in the value of the objective function of a specific optimization model for an incremental change in the variable/equation level.
- `ixmp` (as of v3.3.0) does not store unit attributes for variables and equations.
- In particular models, equations describe specific relationships between data of other types—parameters, variables, and scalars.

The *ixmp* has application programming interfaces (API) for scientific workflows and data processing.

- *Python (ixmp package)*
- *Usage in R via reticulate*
- *Storage back ends (ixmp.backend)*
- *File formats and input/output*

[2] This distinction is also based on the GAMS data model.

- *Mathematical models (ixmp.model)*
- *Reporting / postprocessing*
- *Data model*

CHAPTER

THREE

HELP & REFERENCE

See the index page of the MESSAGEix documentation for a complete overview of different sources of help.

3.1 What's New

3.1.1 Next release

Migration notes

Update code that imports from the following modules:

- `ixmp.reporting` → use `ixmp.report`.
- `ixmp.reporting.computations` → use `ixmp.report.operator`.
- `ixmp.utils` → use `ixmp.util`.

Code that imports from the old locations will continue to work, but will raise `DeprecationWarning`.

All changes

- Support for Python 3.7 is dropped (PR #492).
- Rename *`ixmp.report`* and *`ixmp.util`* (PR #500).
- New reporting operators *`from_url()`*, *`get_ts()`*, and *`remove_ts()`* (PR #500).
- New CLI command **`ixmp platform copy`** and *CLI documentation* (PR #500).
- New argument `indexed_by=...` to *`Scenario.items()`* (thus *`Scenario.par_list()`* and similar methods) to iterate over (or list) only items that are indexed by a particular set (#402, PR #500).
- New *`util.discard_on_error()`* and matching argument to *`TimeSeries.transact()`* to avoid locking *`TimeSeries`* / *`Scenario`* on failed operations with *`JDBCBackend`* (PR #488).
- Work around limitations of *`JDBCBackend`* (PR #500):
  - Unit `""` cannot be added with the Oracle driver (#425).
  - Certain items (variables) could not be initialized when providing `idx_sets=...`, even if those match the sets fixed by the underlying Java code. With this fix, a matching list is silently accepted; a different list raises `NotImplementedError`.
  - When a *`GAMSModel`* is solved with an LP status of 5 (optimal, but with infeasibilities after unscaling), *`JDBCBackend`* would attempt to read the output GDX file and fail, leading to an uninformative error message (#98). Now *`ModelError`* is raised describing the situation.

- Improved type hinting for static typing of code that uses *ixmp* (#465, PR #500).

3.1.2 v3.7.0 (2023-05-17)

All changes

- *ixmp* is tested and compatible with Python 3.11 (PR #481).
- *ixmp* is tested and compatible with pandas 2.0.0 (PR #471). Note that pandas 1.4.0 dropped support for Python 3.7: thus while *ixmp* still supports Python 3.7 this is achieved with pandas 1.3.x, which may not receive further updates (the last patch release was in December 2021). Support for Python 3.7 will be dropped in a future version of *ixmp*, and users are encouraged to upgrade to a newer version of Python.
- Bugfix: *year* argument to *TimeSeries.timeseries()* accepts `int` or `list` of `int` (#440, PR #469).
- Adjust to pandas 1.5.0 (PR #458).
- New module *util.sphinx_linkcode_github* to link documentation to source code on GitHub (PR #459).

3.1.3 v3.6.0 (2022-08-17)

All changes

- Optionally tolerate failures to add individual items in *store_ts()* reporting computation (PR #451); use `timeseries_only=True` in check-out to function with *Scenario* with solution data stored.
- Bugfix: *Config* squashed configuration values read from `config.json`, if the respective keys were registered in downstream packages, e.g. `message_ix`. Allow the values loaded from file to persist (PR #451).
- Adjust to genno 1.12 and set this as the minimum required version for *ixmp.reporting* (PR #451).
- Add *enforce()* to the *Model* API for enforcing structure/data consistency before *Model.run()* (PR #450).

3.1.4 v3.5.0 (2022-05-06)

All changes

- Add new logo and diagram to the documentation (PR #446).
- Raise an informative `ValueError` when adding infinite values with *add_timeseries()*; this is unsupported on *JDBCBackend* when connected to an Oracle database (PR #443, #442).
- New attribute *url* for convenience in constructing *TimeSeries*/*Scenario* URLS (PR #444).
- New *store_ts()* reporting computation for storing time-series data on a *TimeSeries*/*Scenario* (PR #444).
- Improve performance in *add_par()* (PR #441).
- Minimum requirements are increased for dependencies (PR #435):
  - Python 3.7 or greater. Python 3.6 reached end-of-life on 2021-12-31.
  - Pandas 1.2 (2020-12-26) or greater, the oldest version with a minimum Python version of 3.7.
- Improvements to configuration (PR #435):
  - The *jvmargs* argument to *JDBCBackend* can be set via the command line (**`ixmp platform add ...`**) or *Config.add_platform()*; see *Configuration* (#408).

  - Bug fix: user config file values from downstream packages (e.g. `message_ix`) are respected (#415).

- Security: upgrade Log4j to 2.17.1 in Java code underlying *JDBCBackend* to address CVE-2021-44228, a.k.a. "Log4Shell" (PR #445).

  The ixmp Python package is not network-facing *per se* (unless exposed as such by user code; we are not aware of any such applications), so remote code execution attacks are not a significant concern. However, users should still avoid running unknown or untrusted code provided by third parties with versions of ixmp prior to 3.5.0, as such code could be deliberately crafted to exploit the vulnerability.

3.1.5 v3.4.0 (2022-01-24)

Migration notes

`ixmp.util.isscalar()` is deprecated. Code should use `numpy.isscalar()`.

All changes

- Add *TimeSeries.transact()*, for wrapping data manipulations in *check_out()* and *commit()* operations (PR #422).
- Add *Data model*, a documentation page giving a complete description of the *ixmp* data model (PR #422).
- Add the **`pytest --user-config`** command-line option, to use user's local configuration when testing (PR #422).
- Adjust *format_scenario_list()* for changes in `pandas` 1.3.0 (PR #421).

3.1.6 v3.3.0 (2021-05-28)

Migration notes

`rixmp` is deprecated, though not yet removed, as newer versions of the R reticulate package allow direct import and use of the Python modules with full functionality. See the updated page for *Usage in R via reticulate*.

All changes

- Add `ixmp config show` CLI command (PR #416).
- Add `genno` and `message_ix_models` to the output of *show_versions()* / `ixmp show-versions` (PR #416).
- Clean up test suite, improve performance, increase coverage (PR #416).
- Adjust documentation for deprecation of `rixmp` (PR #416).
- Deprecate *util.logger()* (PR #399).
- Add a *quiet* option to *GAMSModel* and use in testing (PR #399).
- Fix *GAMSModel* would try to write GDX data to filenames containing invalid characters on Windows (PR #398).
- Format user-friendly exceptions when GAMSModel errors (#383, PR #398).
- Adjust *ixmp.reporting* to use `genno` (PR #397).
- Fix two minor bugs in reporting (PR #396).

3.1.7 v3.2.0 (2021-01-24)

All changes

- Increase JPype minimum version to 1.2.1 (PR #394).
- Adjust test suite for pandas v1.2.0 (PR #391).
- Raise clearer exceptions from *add_par()* for incorrect parameters; silently handle empty data (PR #374).
- Depend on `openpyxl` instead of `xlrd` and `xlsxwriter` for Excel I/O; `xlrd` versions 2.0.0 and later do not support `.xlsx` (PR #389).
- Add a parameter for exporting all model+scenario run versions to *Platform.export_timeseries_data()*, and fix a bug where exporting all runs happens unintended (PR #367).
- Silence noisy output from ignored exceptions on JDBCBackend/JVM shutdown (PR #378).
- Add a utility method, *gams_version()*, to check the installed version of GAMS (PR #376). The result is displayed by the `ixmp show-versions` CLI command/*show_versions()*.
- *init_par()* and related methods accept any sequence (not merely `list`) of `str` for the *idx_sets* and *idx_names* arguments (PR #376).

3.1.8 v3.1.0 (2020-08-28)

All changes

ixmp v3.1.0 coincides with message_ix v3.1.0.

- Fix a bug in *read_excel()* when parameter data is spread across multiple sheets (PR #345).
- Expand documentation and revise installation instructions (PR #363).
- Raise Python exceptions from *JDBCBackend* (PR #362).
- Add *Scenario.items()*, *util.diff()*, and allow using filters in CLI command `ixmp export` (PR #354).
- Add functionality for storing 'meta' (annotations of model names, scenario names, versions, and some combinations thereof) (PR #353).
  - Add *Backend.add_model_name()*, *add_scenario_name()*, *get_model_names()*, *get_scenario_names()*, *get_meta()*, *set_meta()*, *remove_meta()*.
  - Allow these to be called from *Platform* instances.
  - Remove `Scenario.delete_meta()`.
- Avoid modifying indexers dictionary in `AttrSeries.sel` (PR #349).
- Add region/unit parameters to *Platform.export_timeseries_data()* (PR #343).
- Preserve dtypes of index columns in *data_for_quantity()* (PR #347).
- `ixmp show-versions` includes the path to the default JVM used by JDBCBackend/JPype (PR #339).
- Make `reporting.Quantity` classes interchangeable (PR #317).
- Use GitHub Actions for continuous testing and integration (PR #330).

3.1.9 v3.0.0 (2020-06-05)

ixmp v3.0.0 coincides with message_ix v3.0.0.

Migration notes

Excel input/output (I/O)
The file format used by `Scenario.to_excel()` and `read_excel()` is now fully specified; see *File formats and input/output*.

ixmp writes and reads items with more elements than the ~10^6 row maximum of the Excel data format, by splitting these across multiple sheets.

The I/O code now explicitly checks for situations where the index *sets* and *names* for an item are ambiguous; see *this example* for how to initialize and read these items.

Updated dependencies
The minimum versions of the following dependencies are increased:

- JPype1 0.7.5
- pandas 1.0
- dask 2.14 (for reporting)

Deprecations and deprecation policy
The following items, marked as deprecated in ixmp 2.0, are removed (PR #254):

- `$HOME/.local/ixmp/` as a configuration location. Configuration files are now placed in the standard `$HOME/.local/share/ixmp/`.
- positional and `dbtype=` arguments to `Platform`/`JDBCBackend`.
- `first_model_year=`, `keep_sol=`, and `scen=` arguments to `clone()`. Use *shift_first_model_year*, *keep_solution*, and *scenario*, respectively.
- `rixmp.legacy`, an earlier version of *the R interface* that did not use reticulate.

Newly deprecated is:

- *cache* keyword argument to `Scenario`. Caching is controlled at the `Platform`/Backend level, using the same keyword argument.

Starting with ixmp v3.0, arguments and other features marked as deprecated will follow a standard deprecation policy: they will be removed no sooner than the second major release following the release in which they are marked deprecated. For instance, a feature marked deprecated in ixmp version "10.5" would be retained in ixmp versions "11.x", and removed only in version "12.0" or later.

All changes

- Bump JPype dependency to 0.7.5 (PR #327).
- Improve memory management in `JDBCBackend` (PR #298).
- Raise user-friendly exceptions from `Reporter.get` in Jupyter notebooks and other read–evaluate–print loops (REPLs) (PR #316).
- Ensure `Model.initialize()` is always called for new *and* cloned objects (PR #315).
- Add CLI command *ixmp show-versions* to print ixmp and dependency versions for debugging (PR #320).
- Bulk saving for metadata and exposing documentation AP (PR #314)I

- Add `apply_units()`, `select()` reporting operators; expand `Reporter.add` (PR #312).
- `Reporter.add_product` accepts a `Key` with a tag; `aggregate()` preserves `Quantity` attributes (PR #310).
- Add CLI command `ixmp solve` to run model solver (PR #304).
- Add *dims* and *units* arguments to `Reporter.add_file`; remove `Reporter.read_config()` (redundant with `Reporter.configure`) (PR #303).
- Add option to include *subannual* column in dataframe returned by *`TimeSeries.timeseries()`* (PR #295).
- Add *`Scenario.to_excel()`* and *`read_excel()`*; this functionality is transferred to ixmp from `message_ix` and enhanced for dealing with maximum row limits in Excel (PR #286, PR #297, PR #309).
- Include all tests in the ixmp package (PR #270).
- Add *`Model.initialize()`* API to help populate new Scenarios according to a model scheme (PR #212).
- Apply units to reported quantities (PR #267).
- Increase minimum pandas version to 1.0; adjust for API changes and deprecations (PR #261).
- Add *`export_timeseries_data()`* to write data for multiple scenarios to CSV (PR #243).
- Implement methods to get and create new subannual timeslices (PR #264).

3.1.10 v2.0.0 (2020-01-14)

ixmp v2.0.0 coincides with message_ix v2.0.0.

Migration notes

Support for **Python 2.7 is dropped** as it has reached end-of-life, meaning no further releases will be made even to fix bugs. See PEP-0373 and https://python3statement.org. `ixmp` users must upgrade to Python 3.

Configuration for ixmp and its storage backends has been streamlined. See the ref:*Configuration* section of the documentation for complete details on how to use `ixmp platform add` register local and remote databases. To migrate from pre-2.0 settings:

DB_CONFIG_PATH
…pointed to a directory containing database properties (.properties) files.

- All Platform configuration is stored in one ixmp configuration file, config.json, and manipulated using the `ixmp platform` command and subcommands.
- The *`Platform`* constructor accepts the name of a stored platform configuration.
- Different storage backends may accept relative or absolute paths to backend-specific configuration files.

DEFAULT_DBPROPS_FILE
…gave a default backend via a file path.

- On the command line, use `ixmp platform add default NAME` to set `NAME` as the default platform.
- This platform is loaded when `ixmp.Platform()` is called without any arguments.

DEFAULT_LOCAL_DB_PATH
…pointed to a default *local* database.

- *`ixmp.config`* always contains a platform named 'local' that is located below the configuration path, in the directory 'localdb/default'.
- To change the location for this platform, use e.g.: `ixmp platform add local jdbc hsqldb PATH`.

All changes

- Add `ixmp list` command-line tool (PR #240).
- Ensure filters are always converted to string (PR #225).
- Identify and load Scenarios using URLs (PR #189).
- Add new Backend, Model APIs and CachingBackend, JDBCBackend, GAMSModel classes (PR #182, PR #200, PR #213, PR #217, PR #230, PR #245, PR #246).
- Enhance reporting (PR #188, PR #195).
- Add ability to pass *gams_args* through *solve()* (PR #177).
- Drop support for Python 2.7 (PR #175, PR #239).
- Set *convertStrings=True* for JPype >= 0.7; see the JPype changelog (PR #174).
- Make AppVeyor CI more robust; support pandas 0.25.0 (PR #173).
- Add support for handling geodata (PR #165).
- Fix exposing whole config file to log output (PR #232).

3.1.11 v0.2.0 (2019-06-25)

ixmp 0.2.0 provides full support for *clone()* across platforms (database instances), e.g. from a remote database to a local HSQL database. IAMC-style timeseries data is better supported, and can be used to store processed results, together with model variables and equations.

Other improvements include a new, dedicated *ixmp.testing* module, and user-supplied callbacks in *solve()*. The `retixmp` package using reticulate to access the ixmp API is renamed to `rixmp` and now has its own unit tests (the former `rixmp` package can be accessed as `rixmp.legacy`).

Release 0.2.0 coincides with MESSAGEix release 1.2.0.

All changes

- Test `rixmp` (former `retixmp`) using the R `testthat` package (PR #135).
- Cloning across platforms, better support of IAMC_style timeseries data, preparations for MESSAGEix release 1.2 in Java core (PR #142).
- Support iterating with user-supplied callbacks (PR #115).
- Recognize `IXMP_DATA` environment variable for configuration and local databases (PR #130).
- Fully implement *clone()* across platforms (databases) (PR #129, PR #132).
- New module *ixmp.testing* for reuse of testing utilities (PR #128, PR #137).
- Add functions to view and add regions for IAMC-style timeseries data (PR #125).
- Return absolute path from `find_dbprops()` (PR #123).
- Switch to RTD Sphinx theme (PR #118).
- Bugfix and extend functionality for working with IAMC-style timeseries data (PR #116).
- Add functions to check if a Scenario has an item (set, par, var, equ) (PR #111).
- Generalize the internal functions to format index dimensions for mapping sets and parameters (PR #110).

- Improve documentation (PR #108).
- Replace deprecated pandas `.ix` indexer with `.iloc` (PR #105).
- Specify dependencies in setup.py (PR #103).

3.1.12 v0.1.3 (2018-11-21)

- Connecting to multiple databases, updating MESSAGE-scheme scenario specifications to version 1.1 (PR #88).
- Can now set logging level which is harmonized between Java and Python (PR #80).
- Adding a deprecated-warning for *ixmp.Scenario* with *scheme=='MESSAGE'* (PR #79).
- Changing the API from `mp.Scenario(...)` to `ixmp.Scenario(mp, ...)` (PR #76).
- Adding a function *`has_solution()`*, rename kwargs to *... _solution* (PR #73).
- Bring retixmp available to other users (PR #69).
- Support writing multiple sheets to Excel in utils.pd_write (PR #64).
- Now able to connect to multiple databases (Platforms) (PR #61).
- Add MacOSX support in CI (PR #58).
- Add ability to load all scenario data into memory for fast subsequent computation (PR #52).

3.2 References

3.3 User guidelines and notice

The user guidelines and notice for MESSAGEix also apply to ixmp in their entirety. We ask that you read the notice and take the actions whenever you:

- **use** the *ix modeling platform*, or any model(s) you have built using this tool
- to **produce** any scientific publication, technical report, website, or other **publicly-available material**.

The guidelines include the *optional* step to cite the code via Zenodo (in addition to the *required* step to cite the peer-reviewed publication). You must decide which records to cite:

- If you are using both `message_ix` *and* *`ixmp`*, then cite the MESSAGEix records.
- If you are using *only* *`ixmp`* *without* `message_ix`, then cite the ixmp records. ixmp has its own Zenodo records that are distinct from message_ix:
  - The DOI 10.5281/zenodo.4005665 represents *all* versions of the *`ixmp`* code, and will always resolve to the latest version.
  - At that page, you can also choose a different DOI in order to cite one specific version; for instance, 10.5281/zenodo.4005666 to cite v3.1.0.

3.4 Contributing to development

The MESSAGE*ix* software stack, including *ixmp* and `message_ix`, is developed by a single group of contributors in an integrated process. The MESSAGEix documentation contains complete guidelines for contributing to the code base, at: https://docs.messageix.org/en/latest/contributing.html. All of these guidelines apply to *ixmp*.

Contributors who have signed the contributor license agreement (CLA) for MESSAGE*ix* will need to separately sign the (identical) *CLA for ixmp* when making their first pull request to *ixmp*.

3.4.1 Performance tests

The test suite contains performance tests, which are disabled by the default options given in `setup.cfg`. To run these, use:

```
pytest ... --benchmark-only ...
```

3.5 Contributor License Agreement

3.5.1 Summary and scope

It may seem self-evident that contributing to a project distributed under an open-source license is an implicit permission to anyone for using the contributed code. However, a formal Contributor License Agreements (CLA) makes contribution terms explicit and provides the project maintainers a record of your agreement to those terms.

A wide range of terms exist in other CLAs, including waiver of moral rights, consequential damages waiver, as-is disclaimer, etc. For this project, we follow the more bare-boned GitHub CLA, which focuses on the three most important clauses: copyright, patent, and source of contribution.

In short, by signing this Contributor License Agreement, you confirm that:

1. Anyone can use your contributions anywhere, for free, forever.
2. Your contributions do not infringe on anyone else's rights.

3.5.2 Definition of terms

The following terms are used throughout this agreement:

- **You** - the person or legal entity including its affiliates asked to accept this agreement. An affiliate is any entity that controls or is controlled by the legal entity, or is under common control with it.
- **Project** - the repositories `message_ix` and `ixmp`, and any derived repositories, projects, or software/code packages.
- **Contribution** - any type of work that is submitted to a Project, including any modifications or additions to existing work.
- **Submitted** - conveyed to a Project via a pull request, commit, issue, or any form of electronic, written, or verbal communication with GitHub, contributors or maintainers.

3.5.3 1. Grant of Copyright License

Subject to the terms and conditions of this agreement, You grant to the Projects' maintainers, contributors and users a perpetual, worldwide, unlimited in scope, non-exclusive, no-charge, royalty-free, irrevocable copyright license to, in particular without being limited to, reproduce, prepare derivative works of, publicly display, make available, sublicense, and distribute Your contributions and such derivative works in whole or in part. Except for this license, You reserve all moral rights, title, and interest in your contributions.

3.5.4 2. Grant of Patent License

Subject to the terms and conditions of this agreement, You grant to the Projects' maintainers, contributors and users a perpetual, worldwide, unlimited in scope, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer your contributions, in whole or in part, where such license applies only to those patent claims licensable by you that are necessarily infringed by your contribution or by combination of your contribution with the project to which this contribution was submitted.

If any entity institutes patent litigation - including cross-claim or counterclaim in a lawsuit - against You alleging that your contribution or any project it was submitted to constitutes or is responsible for direct or contributory patent infringement, then any patent licenses granted to that entity under this agreement shall terminate as of the date such litigation is filed.

3.5.5 3. Source of Contribution

Your contribution is either your original creation or based upon previous work that, to the best of your knowledge, is covered under an appropriate open source license. You assure that you are legally entitled to submit your contribution and grant the above license, or you have clearly identified the source of the contribution and any license or other restriction (like related patents, trademarks, and license agreements) of which you are personally aware. If your employer(s) or employee(s) have rights to intellectual property that you create, you represent that you have received permission to make the contributions on behalf of that employer/employee, or that your employer/employee has waived such rights for your contributions.

Should the licensor be held responsible for any violation of intellectual property right in relation to your contribution, you shall be fully liable for damages that may arise.

3.5.6 Reference and License

This Contributor License Agreement and the introductory text is adapted from the GitHub Contributor License Agreement, Version 298f3afd updated August 9, 2017. GitHub granted a CC-BY-4.0 License to IIASA to use and modify the text of the CLA.

CHAPTER

FOUR

LICENSE & USER GUIDELINES

The *ix modeling platform* is licensed under an APACHE 2.0 open-source license.

Please see the *User guidelines and notice* for using the platform in scientific research. *Contributions* to the platform itself are also welcome; new contributors are asked to sign a *Contributor License Agreement*.

BIBLIOGRAPHY

[1] George B. Dantzig. A Transportation Problem. In *Linear Programming and Extensions*, chapter 3.3. Princeton University Press, Princeton, NJ, US, 1963.

[2] Daniel Huppmann, Matthew Gidden, Oliver Fricko, Peter Kolp, Clara Orthofer, Michael Pimmer, Nikolay Kushin, Adriano Vinca, Alessio Mastrucci, Keywan Riahi, and Volker Krey. The MESSAGEix Integrated Assessment Model and the ix modeling platform (ixmp): An open framework for integrated and cross-cutting analysis of energy, climate, the environment, and sustainable development. *Environmental Modelling & Software*, 112:143–156, 2019. doi:10.1016/j.envsoft.2018.11.012.

[3] Clara Orthofer, Daniel Huppmann, and Volker Krey. South Africa's shale gas resources - chance or challenge? *Frontiers in Energy Research*, 2019. doi:10.3389/fenrg.2019.00020.

[4] R.E. Rosenthal. A GAMS Tutorial. In *GAMS: A User's Guide*, chapter 2. The Scientific Press, Redwood City, CA, US, 1988.

PYTHON MODULE INDEX

i

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W