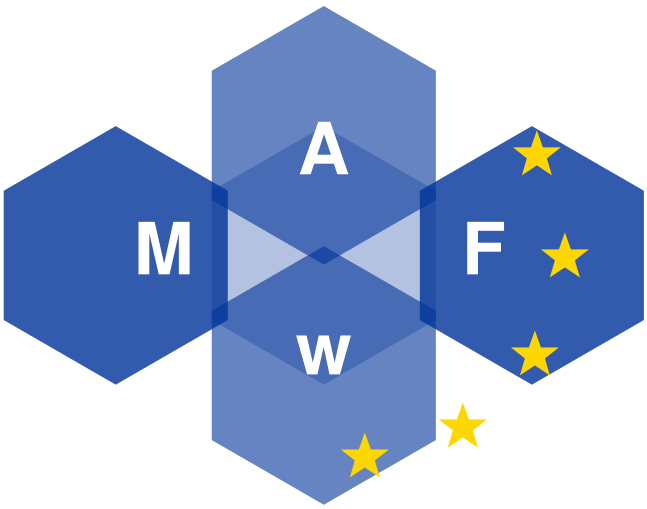


MAFw: Modular Analysis Framework

Release 2.3.0

Antonio Bulgheroni et al.

Aug 31, 2026

CONTENTS:

1 First steps **1**
- 1.1 Installation . . . 1
  - 1.1.1 Avoid problems with source code compilation . . . 1
  - 1.1.2 Requirements . . . 2
- 1.2 Contributing . . . 3
- 1.3 Testing . . . 5
  - 1.3.1 Property based testing . . . 6
  - 1.3.2 Server based database testing . . . 6
- 1.4 Citing MAFw . . . 7

2 Introduction **9**
- 2.1 Statement of need . . . 9
- 2.2 MAFw conceptual design . . . 9
- 2.3 The way ahead . . . 10
  - 2.3.1 Introduce interactivity . . . 10

3 Processor: The core of MAFw **11**
- 3.1 Execution workflow . . . 11
  - 3.1.1 The single loop execution . . . 11
  - 3.1.2 The for loop execution . . . 12
  - 3.1.3 The parallel for loop execution . . . 12
  - 3.1.4 The parallel for loop with queue and batches . . . 13
  - 3.1.5 Loop tuning with steering files . . . 13
  - 3.1.6 The while loop execution . . . 14
  - 3.1.7 How to switch from one loop type to another . . . 14
- 3.2 Subclassing . . . 14
  - 3.2.1 List of methods to be overloaded . . . 15
  - 3.2.2 Customize and document you processor . . . 16
- 3.3 Processor parameters . . . 16
  - 3.3.1 What's behind an `ActiveParameter` . . . 17
  - 3.3.2 Parameter configuration . . . 18
  - 3.3.3 Static parameter schema . . . 19
  - 3.3.4 Saving parameter configuration . . . 19
  - 3.3.5 Parameter and processor inheritance . . . 21
  - 3.3.6 Parameter names . . . 22
  - 3.3.7 Parameter typing . . . 22
- 3.4 What's next . . . 24

4 Processor Examples **25**
- 4.1 Simple and looping processors . . . 25
- 4.2 Modify the *for loop* cycle using the `LoopingStatus` . . . 27
- 4.3 For and while loop execution workflow . . . 29

4.4 Parallel execution . . . 36
4.4.1 Parallel for loop execution . . . 36
4.4.2 Parallel for loop with consumer / producer pattern . . . 37
4.4.3 Which implementation to prefer? . . . 40
4.4.4 Parameter optimization . . . 40
4.5 Importing elements to the database . . . 45
4.5.1 Retrieving information from filenames . . . 45
4.5.2 The basic importer . . . 48
4.5.3 The ImportExample processor . . . 48

5 ProcessorList: Combine your processors in one go **51**
5.1 The `ProcessorExitStatus` . . . 52
5.2 Resources acquisition and distribution . . . 53
5.3 What's next . . . 53

6 Plugins: import your processors **55**
6.1 The quick and dirty: code inside MAFw . . . 55
6.2 The clean and effective: turn your library into a plugin . . . 55
6.2.1 Using a proxy instead of a real processor . . . 57
6.3 What's next . . . 58

7 Run your processors from the command line **59**
7.1 List the processors . . . 59
7.2 Prepare a generic steering file . . . 60
7.2.1 Environment variable expansion . . . 61
7.3 Run the steering file . . . 61
7.4 Debugging your processors . . . 61
7.5 Running more instances of the same processor . . . 62
7.6 Grouping processors together . . . 63
7.7 The exit code . . . 63
7.8 Commands abbreviation . . . 64
7.9 Shell completion . . . 64
7.10 What's next . . . 64

8 Database: your new buddy! **65**
8.1 Database: one name, many implementations . . . 65
8.2 Peewee: a simple and small ORM . . . 65
8.2.1 Database drivers . . . 66
8.3 One class for each table . . . 66
8.4 Ready, go! . . . 69
8.4.1 Configuring other types of databases . . . 70
8.5 The advantages of using MAFwBaseModel . . . 72
8.5.1 Automatic join helper . . . 72
8.5.2 Automatic model registration . . . 72
8.5.3 Trigger support . . . 74
8.5.4 Customizable table name . . . 74
8.5.5 Automatic table creation . . . 75
8.6 Triggers: when the database works on its own . . . 75
8.6.1 The signal approach . . . 76
8.6.2 The trigger approach . . . 78
8.6.3 Triggers on different databases . . . 82
8.6.4 Debugging triggers . . . 84
8.7 Standard tables . . . 84
8.7.1 Default standard tables . . . 85
8.8 Custom fields . . . 85
8.8.1 Removing widow rows . . . 86
8.8.2 Pruning orphan files . . . 86
8.8.3 Keeping the entries updated . . . 90
8.9 Multi-primary key columns . . . 91

8.10 Importing an existing DB . . . 94
8.11 Execute SQL scripts . . . 95
8.12 What's next . . . 96

9 Filters: do only what is needed to be done! 97
9.1 How to use a filter . . . 97
9.2 How to configure a filter . . . 101
9.3 Explicit operation configuration . . . 102
9.4 Logical Composition . . . 103
9.5 Conditional filters . . . 104
9.6 Filter schema metadata, your help to configure filters . . . 105
9.6.1 Disabling a filter . . . 106
9.7 How to use the *new_only* filter . . . 106
9.7.1 Basic usage example . . . 106
9.7.2 New only flag and processor parameters . . . 106
9.7.3 Advanced *new_only* filtering with multiple column primary keys . . . 107
9.8 Why do I get an error about model binging? . . . 108
9.9 Filter migration from MAFw v1.4.0 . . . 109
9.10 What's next . . . 109

10 Display your results 111
10.1 Add diagnostic information to your processors . . . 111
10.2 Generate high level graphs . . . 112
10.2.1 Getting the data . . . 113
10.2.2 Plotting the data . . . 114
10.2.3 Mixin approach . . . 115

11 A step by step tutorial for a real experimental case study 119
11.1 The experimental scenario . . . 119
11.1.1 Task 0. Generating the data . . . 119
11.1.2 Task 1. Building your data bank . . . 120
11.1.3 Task 2. Do the analysis . . . 120
11.1.4 Task 3. Prepare a relation plot . . . 120
11.2 The 'code' . . . 120
11.2.1 The database definition . . . 121
11.2.2 The processor library . . . 124
11.2.3 The plugin module . . . 128
11.3 Run the code! . . . 128

12 Steering Files 133
12.1 Steering GUI User Guide . . . 133
12.1.1 Why use the Steering GUI? . . . 133
12.1.2 Installation . . . 133
12.1.3 General Operation . . . 134
12.1.4 The Editors . . . 134
12.2 The backbone of the GUI: the Steering Builder . . . 140
12.2.1 Overview . . . 140
12.2.2 Builder API . . . 141
12.2.3 Serialization & Validation . . . 141

13 Development tools 143
13.1 Requirements . . . 143
13.2 How to invoke `devtools` . . . 143
13.3 Command structure . . . 144
13.4 Shell completion . . . 144
13.5 Documentation commands . . . 145
13.5.1 Purpose and idea . . . 145
13.5.2 Typical usage examples . . . 145
13.6 Release commands . . . 146

13.6.1 Purpose and idea . . . 146
13.6.2 Release reproducibility and rolling compatibility . . . 146
13.6.3 Documentation target version . . . 146
13.6.4 Prepare a new release (`release create`) . . . 146
13.7 Dependencies commands . . . 146
13.7.1 Purpose and idea . . . 146
13.7.2 Verify dependency stacks . . . 147
13.7.3 Compare dependency versions (`dependencies latest compare`) . . . 147
13.7.4 Freeze and unfreeze . . . 147
13.7.5 Manage dependency reference files (`dependencies registry`) . . . 147
13.7.6 Vulnerabilities check (`dependencies audit`) . . . 147
13.8 Toolchain commands . . . 147
13.8.1 Purpose and idea . . . 147
13.8.2 Command tree . . . 148
13.8.3 Group-level options . . . 148
13.8.4 Subcommand-level filtering options . . . 148
13.8.5 List (`toolchain list`) . . . 148
13.8.6 Check (`toolchain check`) . . . 149
13.8.7 Update (`toolchain update`) . . . 149
13.8.8 Verify (`toolchain verify`) . . . 149
13.8.9 Bootstrap (`toolchain bootstrap`) . . . 149
13.8.10 Managed tools . . . 149

14 API **151**
14.1 mafw . . . 151
14.1.1 mafw.active . . . 152
14.1.2 mafw.db . . . 153
14.1.3 mafw.decorators . . . 190
14.1.4 mafw.devtools . . . 194
14.1.5 mafw.enumerators . . . 321
14.1.6 mafw.examples . . . 323
14.1.7 mafw.hookspecs . . . 337
14.1.8 mafw.lazy_import . . . 337
14.1.9 mafw.mafw_errors . . . 340
14.1.10 mafw.models . . . 344
14.1.11 mafw.plugin_manager . . . 347
14.1.12 mafw.plugins . . . 351
14.1.13 mafw.processor . . . 351
14.1.14 mafw.processor_library . . . 387
14.1.15 mafw.runner . . . 414
14.1.16 mafw.scripts . . . 416
14.1.17 mafw.steering . . . 426
14.1.18 mafw.steering_gui . . . 434
14.1.19 mafw.timer . . . 474
14.1.20 mafw.tools . . . 476
14.1.21 mafw.ui . . . 493

Python Module Index **501**

Index **503**

CHAPTER 1

FIRST STEPS

1.1 Installation

MAFw can be installed using pip in a separated virtual environment.

Windows

```
c:\> python -m venv mafw-env
c:\> cd mafw-env
c:\mafw-env> Scripts\activate
(mafw-env)c:\mafw-env> pip install mafw
```

linux & MacOS

```
$ python -m venv mafw-env
$ cd mafw-env
$ source bin/activate
(mafw-env) $ pip install mafw
```

1.1.1 Avoid problems with source code compilation

MAFw is pure python code, so when you install it using pip, the source code will be copied on your system and interpreted from there. But some of MAFw dependencies require C compilation, for example `matplotlib`.

If during the installation you see an error message indicating a failed attempt to built a package from source, likely due to a missing compiler chain, then you need to instruct pip to rely only on binary packages. MAFw requirements compatibility range is carefully selected in a way to avoid this situation, because we want to keep the installation as easy as possible, but we cannot exclude it completely especially in the case your are using an old - but still supported - python version. Should this happens, no fear, just replace your standard pip install command with this one:

```
# Prefer binary packages over source packages, even if the source packages are newer.
(mafw-env) $ pip install --prefer-binary mafw
```

By default, pip is trying to resolve the dependency challenge selecting the newest release of all packages compatible with all the requirements. Adding this additional flag, pip will change its resolution strategy and discard all source-only available packages.

In the unlikely situation that this is also failing, then please submit a bug report with a precise description of your environment (operating system, python version) and a copy of the pip error message.

1.1.2 Requirements

MAFw has been developed using python version 3.11 and tested with newer versions up to the current stable release (3.14). Apart from some *typing* issues, we do not expect problems when running it with older releases. It is our intention to follow the future advancement of python and possibly use the NO-GIL option starting from version 3.14 to improve the overall performances.

Concerning dependencies, all packages required by MAFw are specified in the pyproject file and will be automatically installed by pip. Nevertheless, if you are curious to know what comes with MAFw, here is a list of direct dependencies with the indication of what their role is inside the library.

Dependency	Minimum supported version	Description
pluggy	>=1.5.0, <2; python_version < "3.14"	to implement the plugin mechanism and let the users develop their own processors
	>=1.6.0, <2; python_version >= "3.14"	
click	>=8.3.3, <9	to implement the command line interface for the mafw execution engine
tomlkit	>=0.15.0, <0.16	to implement the reading and writing of steering files
rich	>=13.9.4, <16	to improve the user experience with the console output
peewee	>=4.1.2, <5	to implement the ORM database interface
Deprecated	>=1.2.18, <2	to inform the user about deprecated usages
rich-pyfiglet	>=1.0.0, <3	to display welcome banners
typing-extensions	>=4.13.0, <5; python_version <= "3.11"	to have access to typing annotations

Minimum supported dependency versions are selected to ensure reliable installation using precompiled wheels on all supported Python versions whenever possible. Older versions that require source compilation may still work, but are outside the tested installation matrix.

If MAFw is installed with the additional features provided with by **seaborn**, then those packages will also be installed.

Dependency	Minimum supported version	Description
seaborn	>=0.13.2, <0.14	to implement the generation of high level graphical outputs
pandas[hdf5]	>=2.2.3, <4; python_version < "3.14"	to allow the use of dataframes for data manipulations
	>=2.3.3, <4; python_version >= "3.14"	
matplotlib	>=3.10.1, <4; python_version < "3.14"	the low level graphics interface
	>=3.10.5, <4; python_version >= "3.14"	

By default MAFw comes with an abstract plotting interface. If you want to use seaborn, then just install the optional dependency *pip install mafw[seaborn]*

If you also want to install the GUI for the generation of steering files, then **PySide6** will also be installed. In this case be sure that you OS system is having all the required libraries (on linux mainly *libgl1*).

All MAFw dependencies will be automatically installed by pip.

Parallel processing

MAFw is a scientific library dealing with data analysis and as such it is expected to run as fast and as efficient as possible. Python is known not to be the fastest in the scene of programming and scripting language, but its simplicity and large number of available (optimized!) libraries are giving it a top player role despite its reduced speed.

Since a few years already, CPython has started a difficult migration process to get rid of one big obstacle on the speed race, that is the GIL. In 2024 with python 3.13 the so-called free-threading features were still experimental and two releases were created, one with and another without GIL. With python 3.14 the free-threading option is not anymore experimental and in the period 2026-2027 it is expected that there will be one single python binary with the possibility to switch on and off the GIL at runtime (enabled by default). In 2028-2030, if everything goes as expected, new releases of python will come with the GIL disabled by default [PEP-703]

MAFw supported almost from its beginning the cause of achieving higher performance via multi-threading, in particular because Processors are very often running internal loops and thus benefiting a lot from the possibility to assign each loop item to a different thread. You will learn more about parallel for loops in a dedicated *section*.

Here is a list of known limitations, you should be aware of when using MAFw in free-threading mode:

1. **Python >= 3.14**. MAFw is expected to give reliable results, so the authors waited for the first official release of the free-threading version before embarking into this challenging evolution. This means that MAFw parallel processing was never tested with python ~ 3.13. If you want to give it a try, be aware it was never tested in that environment.
2. **Small UI related bug in Python <= 3.14.3**. There is a small bug in python <= 3.14.3t. It is nothing serious, it will only affect the way warning messages are displayed in the rich ui. A small *guard* has been included in MAFw to protect against this eventuality, so your code will run smoothly also in this case, just with a little bit messier output in case of warnings. The good news is that the bug has been already fixed and included in version 3.14.4, so unless you are still using one of the three affected micro releases, otherwise you are absolutely fine.
3. **Missing libraries**. Some of the libraries used by MAFw are based on *binaries*. It means that you do not have to compile them by your own and pip/hatch/uv will install the pre-built packages making your life easier. At the time of writing (March 2026), PySide6 does not have a pre-built installation package that is compatible with 3.14t. This means that you won't be able to use the steering-gui tool to generate your steering files. You can in any case install MAFw in two environments, one to operate steering-gui and one with free-threading where you execute mafw at full speed. Much more relevant is the missing binary installation package for the PostgreSQL driver (both version 2 and 3). So if you want to use PostgreSQL with free-threading and you do not want to build the pyscopg package from source, you will have to go for the pure python installation that is much slower, not really very smart.
4. **No in-memory databases**. MySQL and Sqlite are good to go in multi-threaded loop (in theory also PostgreSQL, if you compile the library by yourself). There is only one limitation, you cannot use *:memory:* Sqlite database when working with multi-threads. Each thread will create its own copy of the memory database resulting in a fresh new database everytime a thread is starting the job. In our opinion this should not affect much your analysis strategy because it is quite unusual to have real production databases operating in memory.

1.2 Contributing

Contributions to the software development are very much welcome. A more detailed guide on how to contribute or to get help for the development of your processors can be found in CONTRIBUTING.md.

If you plan to contribute to MAFw itself (documentation, code, CI/CD), please also read the *Development tools* section, which explains the helper commands used by the maintainers.

If you want to join the developer efforts, the best way is to clone/fork this repository on your system and start working.

The development team has adopted hatch for basic tasks. So, once you have downloaded the git repository to your system, open a shell there and type:

Windows

```
D:\mafw> hatch env create dev
D:\mafw> hatch env find dev
C:\path\to\.venv\mafw\KVhWIDtq\dev.py3.11
C:\path\to\.venv\mafw\KVhWIDtq\dev.py3.12
C:\path\to\.venv\mafw\KVhWIDtq\dev.py3.13
```

linux & MacOS

```
$ hatch env create dev
$ hatch env find dev
/path/to/venv/dev.py3.11
/path/to/venv/dev.py3.12
/path/to/venv/dev.py3.13
```

to generate the python environments for the development. This command will actually create the whole environment matrix, that means one environment for each supported python version. If you intend to work primarily with one single python version, simply specify it in the create command, for example:

Windows

```
D:\mafw> hatch env create dev.py3.13
D:\mafw> hatch env find dev.py3.13
C:\path\to\.venv\mafw\KVhWIDtq\dev.py3.13
```

linux & MacOS

```
$ hatch env create dev.py3.13
$ hatch env find dev.py3.13
/path/to/venv/dev.py3.13
```

hatch will take care of installing MAFw in development mode with all the required dependencies. Use the output of the find command, if you want to add the same virtual environment to your favorite IDE. Once done, you can spawn a shell in the development environment just by typing:

Windows

```
D:\mafw> hatch shell dev.py3.13
(dev.py3.13) D:\mafw>
```

linux & MacOS

```
$ hatch shell dev.py3.13
(dev.py3.13) $
```

and from there you can simply run mafw and all other scripts.

> **Note**
>
> If you are not familiar **hatch**, we strongly encorage to have a look at their website. This powerful tool, similar to poetry and uv, it is simplifying the creation of a matrix of virtual environments for developing and testing your code with different combination of requirements.

We strongly recommend to install hatch via pipx, so to have the executable available systemwide, but nevertheless running in a separate environment.

MAFw uses pre-commit to assure a high quality code style. The pre-commit package will be automatically installed into your environment, but it must be initialised before its first use. To do this, simply run the following command:

Windows

```
(dev.py3.13) D:\mafw> pre-commit install
```

linux & MacOS

```
(dev.py3.13) $ pre-commit install
```

And now you are really ready to go with your coding!

Before pushing all your commits to the remote branch, we encourage you to run the pre-push tests to be sure that everything still works as expected. You can do this by typing:

Windows

```
D:\mafw> hatch run dev.py3.13:pre-push
```

linux & MacOS

```
$ hatch run dev.py3.13:pre-push
```

if you are not in an activated development shell, or

Windows

```
(dev.py3.13) D:\mafw> hatch run pre-push
```

linux & MacOS

```
(dev.py3.13) $ hatch run pre-push
```

if you are already in the dev environment.

These pre-push checks will include some cosmetic aspects (ruff check and format) and more relevant points like static type checking with mypy, documentation generation with sphinx and functionality tests with pytest.

1.3 Testing

MAFw comes with an extensive unit test suite of more than 1000 test cases for an overall code coverage of 99%.

Tests have been coded using pytest best practice and are aiming to prove the functionality of each unit of code taken individually. Given the high level of interoperability of MAFw with other libraries (toml, peewee and seaborn just to name a few), unit tests rely heavily on patched objects to assure reproducibility.

Nevertheless full integration tests are also included in the test suite. These tests will cover all relevant aspects of MAFw, including:

1. Installation of MAFw and of a Plugin project in a isolated environment
2. Use of MAFw executable to create some data files and analyse them to create a graphical output.
3. Use of a database to store the collected data.

4. Check the database trigger functionalities to avoid repeating useless analysis steps, for example when a new file is added, removed or changed.

If you plan to collaborate in the development of MAFw, you must include unit tests for your contributions.

As already mentioned, MAFw is using hatch as project management. In the pyproject.toml file, hatch is configured to have a matrix of test environment in order to run the whole test suite with the supported versions of python (3.11, 3.12, 3.13 and 3.14).

Running the suite is very easy. Navigate to the folder where you have your local copy of MAFw and type `hatch test`. Hatch will take care of installing the proper environment and run the tests. Should one or more test(s) fail, then the slow integration tests will be skipped to spare some time.

Have a look at the hatch test options, in particular the *-a*, to test over all the environments in the matrix and the *-c* to generate coverage data for the production of a coverage report.

1.3.1 Property based testing

Unit-tests are meant to be fast and to focus on one single unit at the time to properly identify failures. Starting from MAFw 2.2, we have introduced *property-based-testing* using hypothesis plugin. Because of this some tests might be looking taking much more time than expected, but the reason behind is not a bad coding rather a parametrised execution where hundreds of different inputs are validated instead of one. This parametric execution is totally transparent for the user, you just need to be a bit more patient while the test suite is executed.

1.3.2 Server based database testing

MAFw uses peewee as an interface to databases and this library is offering an abstract layer to connect to different type of databases without requiring the user to adapt the code. In other words, you can switch from one database backend to another just be editing the configuration file.

Testing if peewee works is out of the scope of MAFw test-suite, but since MAFw implements a few improvements on top of peewee basic functionalities (for example triggers), these are subject to verification. If these extra functionalities are SQL dialect specific, then we should test that all supported DB backends (Sqlite, PostgreSQL and MySQL) are all behaving as expected and here come the problems. In fact, testing against Sqlite is very easy since the DB is a file if not a memory stored object, but testing against PostgreSQL and MySQL requires a running instance of these database servers.

If you are writing tests for server based DB backends, you need to label your test with *db* mark. This will assure the proper behaviour, meaning that pytest will try to run them if a valid backend is found, otherwise they will be skipped. From the CI environment, db labelled tests will be executed in a separate unit-test job (*db_test*), where additional DB service containers are started.

If you want to test locally your DB development, you need to start those containers on your development machine as well, and we suggest to follow this procedure:

1. In the tests folder there are two examples of environment files (.mysql.env.example and .postgresql.env.example). Rename them to remove the .example suffix and adjust your desired settings.
2. Start the servers using docker or podman (*docker compose up -d*). Adjust the value of the variables in the suggested docker compose file to what you put in the env files.

```
services:
  # MySQL Configuration
  mysql-db:
    image: mysql:latest
    container_name: mysql-server
    restart: always
    environment:
      MYSQL_ROOT_PASSWORD: <ROOT_PASSWORD>
      MYSQL_DATABASE: mafw_db
      MYSQL_USER: <USER_NAME>
      MYSQL_PASSWORD: <PASSWORD>
```

(continues on next page)

(continued from previous page)

```
    ports:
      - "3306:3306"
    volumes:
      - mysql_data:/var/lib/mysql:Z
    command: --log_bin_trust_function_creators=1


  # PostgreSQL Configuration
  postgres-db:
    image: postgres:latest
    container_name: postgres-server
    restart: always
    environment:
      POSTGRES_USER: <USER_NAME>
      POSTGRES_PASSWORD: <PASSWORD>
      POSTGRES_DB: mafw_db
    ports:
      - "5432:5432"
    volumes:
      - pg_data:/var/lib/postgresql/data:Z

volumes:
  mysql_data:
  pg_data:
```

3. Run the dedicated test suite with *hatch -e db-test run pytest -m db -v*
4. Tear down the containers with *docker compose down*

If you still need some guidance, have a look at how tests in *tests/test_db_backends.py* are implemented.

1.4 Citing MAFw

If you used MAFw in your research and you would like to acknoledge the project in your academic publication we suggest citing the following paper:

- Bulgheroni et al., (2025). MAFw: A Modular Analysis Framework for Streamlining and Optimizing Data Analysis Workflows. Journal of Open Source Software, 10(114), 8449, https://doi.org/10.21105/joss.08449

or as BibTeX format:

```
@article{Bulgheroni2025,
    doi = {10.21105/joss.08449},
    url = {https://doi.org/10.21105/joss.08449},
    year = {2025},
    publisher = {The Open Journal},
    volume = {10},
    number = {114},
    pages = {8449},
    author = {Bulgheroni, Antonio and Krachler, Michael},
    title = {MAFw: A Modular Analysis Framework for Streamlining and Optimizing Data␣
↪Analysis Workflows},
    journal = {Journal of Open Source Software}
}
```

Tip

Reading Path Recommendation: This documentation follows a systematic approach, describing all the elements composing MAFw and their interactions before presenting a comprehensive practical example. However, we recognize that different users have varying learning preferences. If you prefer to begin with a concrete implementation to understand the library's capabilities before diving into theoretical foundations, you may jump directly to the *tutorial section*. After reviewing the practical application, you can return to this detailed documentation for a thorough understanding of the underlying concepts and architectural design.

CHAPTER 2

INTRODUCTION

2.1 Statement of need

MAFw addresses the need for a flexible and modular framework that enables data scientists to implement complex analytical tasks in a well-defined environment. Currently, data analysis workflows often require scientists to handle multiple tasks, such as data ingestion, processing, and visualization, which can be time-consuming and prone to errors. Moreover, the lack of standardization in data analysis pipelines can lead to difficulties in reproducing and sharing results.

MAFw aims to fill this gap by providing a Python-based tool that allows data scientists to focus on the analysis itself, rather than on the ancillary tasks. The framework is designed to be highly customizable, enabling users to create their own processors and integrate them into the workflow. A key feature of MAFw is its strong collaboration with a relational database structure, which simplifies the analysis workflow by providing a centralized location for storing and retrieving data. This database integration enables seamless data exchange between different processors, making it easier to manage complex data pipelines.

2.2 MAFw conceptual design

The concept behind MAFw is certainly not novel. Its functionality is so prevalent in data analysis that numerous developers, particularly data scientists, have attempted to create libraries with similar capabilities. MAFw's developers got inspired by MARLIN: this object C++ oriented application framework, no longer being maintained, was offering a modular environment where particle physicists were developing their code in the form of shared libraries that could be loaded at run time in a plugin-like manner[1]. One of MARLIN strengths was the strong connection with the serial I/O persistency data model offered by LCIO.

Starting from those solid foundations, MAFw moved from C++ to python in order to facilitate the on-boarding of data scientists and to profit from the vast availability of analytical tools, replacing the obsolete LCIO backend with a more flexible database supported input/output able to deal with large amount of data with categorical variables without severely impacting on I/O performance.

The general concept behind MAFw has been developed by the authors to perform image analysis on autoradiography images, featuring an ultra simplified database interface (sqlite only) along with some dedicated processors targeting autoradiography specific tasks.

Having understood the potentiality of this scheme, the authors decided to extract the core functionalities of the framework itself, expand the database interface making use of an ORM approach (peewee), include a plugin system to simplify the integration of processors developed for different purposes in external projects and supply an extensive general and API documentation, before releasing the code to the public domain as open source.

[1] One of MAFw developer was in charge of the original coding of EUTelescope, a set of MARLIN processors for the reconstruction of particle trajectories recorded with beam telescopes.

2.3 The way ahead

The future development of MAFw is driven by code usability. The authors are trying their best to make the framework as functional as possible offering colleague scientists a platform where to perform their analyses. At the time of writing there is already one target envisaged: improved interactivity.

2.3.1 Introduce interactivity

Even though in the original implementation of MAFw precursor, interactive processors were already existing, they were temporary removed from the current implementation. The authors recognize that many data scientists prefer to conduct interactive analysis using jupyter or marimo notebooks. Therefore, they are actively exploring ways to seamlessly integrate interactivity into the processor workflow through these notebook environments.

CHAPTER 3

PROCESSOR: THE CORE OF MAFW

The Processor is responsible to carry out a specific analytical task, in other simple words, it takes some input data, it does some calculations or manipulations, and it produces some output data.

The input and output data can be of any type: a simple list of numbers, a structured data frame, a path to a file where data are stored, an output graph or a link to a web resource, a database table, and so on.

In ultra simplified words, a Processor does three sequential actions:

1. Prepare the conditions to operate (`start()`)
2. Process the data (`process()`)
3. Clean up what is left (`finish()`).

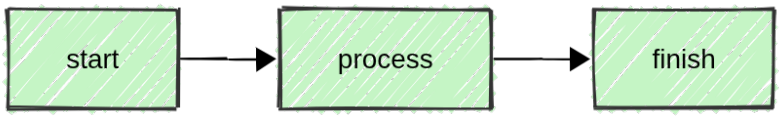
3.1 Execution workflow

There are instances where tasks can be executed in a single operation across your entire dataset, such as generating a graph from the data. However, there are also situations where you need to iterate over a list of similar items, applying the same process repeatedly. Additionally, you may need to continue a process until a specific condition is met. The `Processor` can actually accomplish all those different execution workflows just by changing a variable: `the loop type`.

You will be presented a more detailed description of the different workflows in the following sections.

3.1.1 The single loop execution

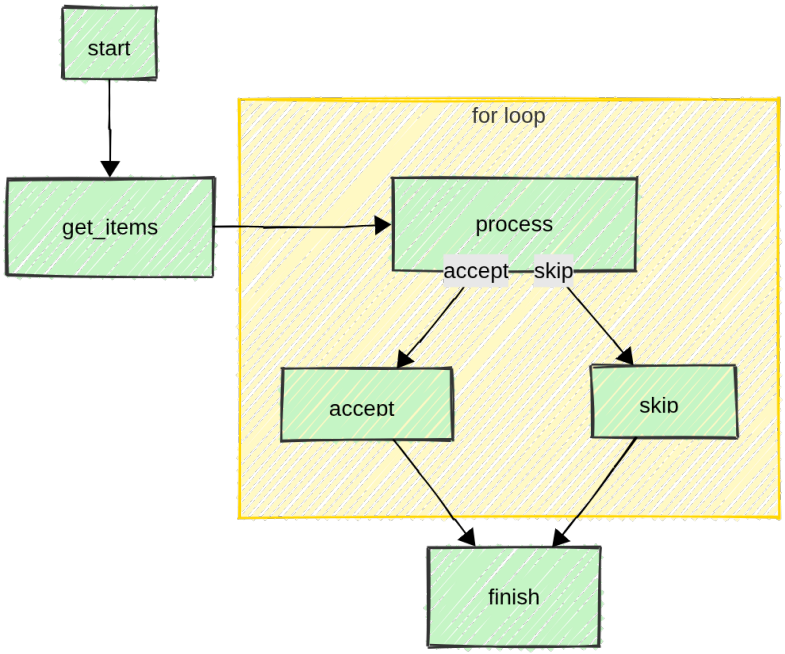
The first family has a simplified execution workflow schematically shown here below:



The distinction of roles between the three methods is purely academic, one could implement all the preparation, the real calculation and the clean up in one method and the processor will work in the same way. The methods are anyhow preserved to offer a similar execution scheme also for the other looping scheme.

3.1.2 The for loop execution

In python a for loop is generally performed on a list of items and MAFw is actually following the same strategy. Here below is the schematic workflow.



As you can see, after having called the `start()`, the user must provide a list of items to be processed implementing the `get_items()`. This can be the list of files in a directory, or the rows in a DB table or even a list of simple numbers; whatever best suit the user's needs. You will soon learn how to deal with *database* entries and how to *filter* them.

Now everything is ready to start the loop and call the `process()` as many times as the items in the input list. In the process implementation, the user can realize that something went wrong with a particular item and can modify what is executed next (`accept_item()` or `skip_item()`). See an example of such a possibility *here*. At the end of the loop it is time to clean up everything, saving files, updating DBs and so one. This is again the task of the `finish()` method.

3.1.3 The parallel for loop execution

The *parallel for loop* follows the same workflow as the standard for loop, but distributes items to a thread pool. **This workflow is available only when Python is running in free-threading mode**. If free-threading is not available, MAFw automatically falls back to the serial for loop and emits a warning.

Parallel execution does **not** guarantee item processing order. The `accept_item()` and `skip_item()` callbacks are executed in the worker threads, so any shared state (for instance lists, counters, or database writes) must be protected accordingly.

For the scientists developing their own processors, it is absolutely transparent if the for loop is executed serially or in a multi-thread pool. Just implement your analysis procedure in the three standard methods (`processor.Processor.start()`, `processor.Processor.process()` and `processor.Processor.finish()`) and MAFw will take care of the rest.

If your processor is generating some results that you want to store locally in an attribute before putting them in a file or in a database table, follow the advice of the free-threading python guide and use a threading.Lock to prevent concurrent modifications even if built in types are supposed to use internal locks.

If you want to access the database from within the loop, the connection is also thread-safe according to peewee documentation.

The number of threads in the executor pool is a parameter that you can modify via the class attribute `max_workers`, by default it is set to the number of cores in your PC plus 4 capped to a maximum of 32. If you want to change this value, just pass *max_workers* to your processor constructor.

Note

You might wonder how the use of class attributes like *processor.Processor.item*, *processor.Processor.i_item* and *processor.Processor.n_item* is possible in the *processor.Processor.process()* implementation when multiple threads are working in parallel. The answer is that MAFw is transforming them in thread-locals before submitting the worker to the executor. If you are curious to know this is technically achieved look at the documentation and the source code of *processor.Processor._thread_loop_context()*.

3.1.4 The parallel for loop with queue and batches

From a pure parallel execution point of view, the locking mechanism inside the process might be a bottle neck in the performance gain and for this reason, MAFw is also offering a producer/consumer pattern implementation of multi-threaded looping. The idea is to have a queue shared among a consumer thread and several producer threads. The consumer threads is taking care of emptying the queue as soon as there are items appearing, while the many producers are just enqueuing processed output as soon as they are ready. A part from the maximum number of workers, for this type of parallel architecture you can also tune other two parameters: the *processor.Processor.queue_size* (by default twice as large as the number of producer threads) and the *processor.Processor.queue_batch_size* (default 1). A bigger queue can cause a large memory spike depending on the size of the payload that the producers are transferring to the consumer, while a smaller queue can cause congestion. In fact when the queue is full (because either the producers are too fast or the consumer is too slow), the whole process is kept on hold and the operating system will pause the producer threads to let the consumer make some space in the queue. If the producers are too fast, meaning that the queue is immediately full, you can group the input items in batches, so that the producers have to perform their calculation on multiple items before transferring a list of results to the queue. A good tuning of the batch size can be a game changer in terms of performance!

How do you have to implement your processor? The queue-based parallel for loop keeps the processing workload in multiple worker threads but funnels the post-processing steps through a single consumer thread. In this workflow, *accept_item()* and *skip_item()* are executed by the consumer thread, providing serialized updates to shared state without explicit locking. Users can customize the consumer behavior by overloading *consumer_start()*, *consumer_process()*, or *consumer_finish()*. In general your process implementation should return (normally it does not return any value) the payload that the consumer will have to retrieve from the queue for post-processing. You just return the value that you want to have in the *consumer_process()* and MAFw will do all the multi-thread plumbing behind the scene to let you receive the value exactly where you need it.

Do you want to see an example? We have you covered, have a look at this *specific example*!

3.1.5 Loop tuning with steering files

I can see the question arising in your mind? How do I know which parallel processing approach should I follow and which are the best performing parameters? Even though there are general rules that will guide your choice in the right direction, very often it is much better to just give it a try. We have seen that the different implementations of the parallel loops come with a bunch of parameters that are not settable via the steering file, but you can pass them via the processor constructor.

This is not particularly handy if you want to explore the parameter space to identify the best performance corner. At the same time, we did not think it was a good idea to pollute each and every processor subclass with additional operational *processor.ActiveParameter* in the steering file. But do not worry, MAFw is always here to help you, and we have found a nice way to 'temporary' include these parameters in your steering file without having to change your processor code. Just include this *LoopTuningMixin* mixin class in your processor subclass (**before** Processor) and the magic will happen.

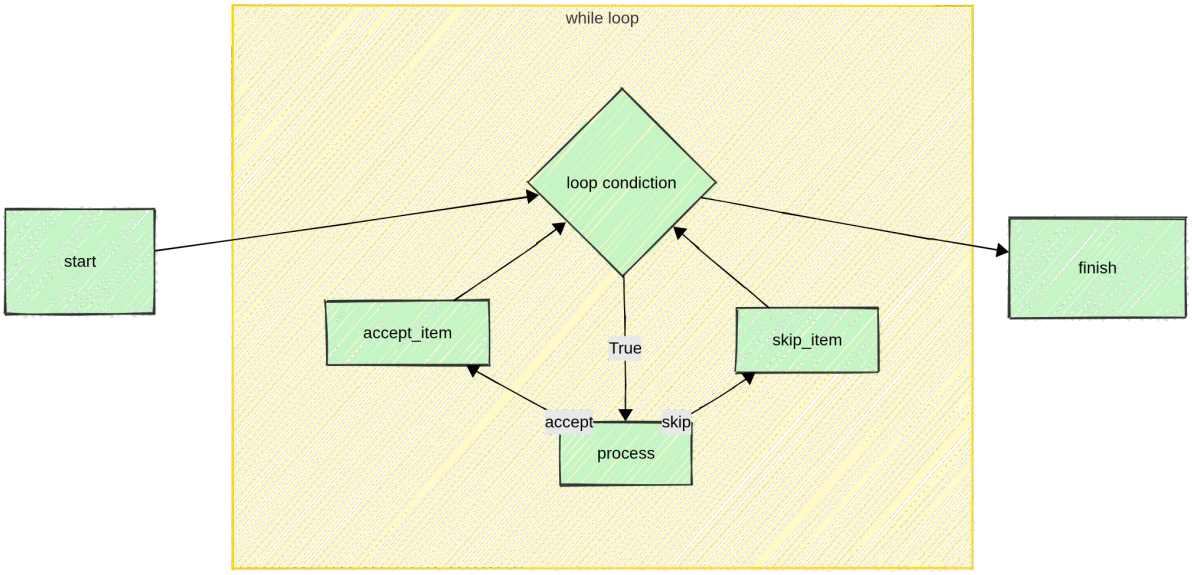
When the mixin is used, the following parameters become available in the steering file:

- `loop_type` (allowed values: `for_loop`, `parallel_for_loop`, `parallel_for_loop_with_queue`)
- `max_workers`
- `queue_size`
- `queue_batch_size`

The intended workflow is to temporarily add the mixin to your processor class, tune the values in the steering file, and once you have found the optimal configuration remove the mixin and use the optimal values in the constructor. If you want, you can also leave the mixin in place, it won't make any hurt.

3.1.6 The while loop execution

From a programming point of view, the for loop and the while loop execution are rather similar. Between the execution of the start and the finish method, the process is repeated for a certain number of times until a certain condition is met.



In this case, there is no list of items to loop over, but a condition that should be checked. Thus the user has to overload the `while_condition()` method to return a boolean value: True if the loop has to continue or False if it has to stop and go to the finish.

3.1.7 How to switch from one loop type to another

It is extremely simple to switch from one execution scheme to another. The `Processor` class takes the argument `loop_type`, just change this value in the processor init and the rest will come automatically.

Remember that, by default a processor is set to work with a *for loop* workflow, and thus you have to implement the `get_items()`. If you switch to *while loop*, then you need to implement `while_condition()` for the system to work.

The *parallel for loop* workflow still requires `get_items()`, but the loop type should be set to `ParallelForLoop` (or using the `parallel_for_loop()` decorator). The *parallel for loop with queue* workflow uses `ParallelForLoopWithQueue` and the `parallel_for_loop_with_queue()` decorator.

The last thing, you need to know is how to run a processor. First you create an instance of your processor, then you call the `execute()` method. There are other more practical ways, not involving any coding that we will discuss *later on*.

A comparison between the **for loop** and the **while loop** execution workflow is described in the *example page*.

3.2 Subclassing

The basic processor provided in the library does nothing specifically, it is only used as a skeleton for the execution. In order to perform some real analysis, the user has to subclass it and overload some methods. Having gained a clear understanding of the role of each steps in the execution workflow, read the list of methods that can be overloaded here below and then you are ready to see some simple *Processor Examples*.

3.2.1 List of methods to be overloaded

The *process()* method.

> This is the central part of the processor, and it must contain all the calculations. If the processor is looping on a list of input items, the user can access the current item via the *item* attribute.

The *start()* method.

> This is useful to prepare the condition to operate. For example, it is a good idea to open files and do all the preparatory work. In a looping processor, this method is called just before the cycle starts.

The *finish()* method.

> This is useful to clean up after the work. For example, to save files, update DB tables and so on. In a looping processor this is performed just after the last iteration. Here is also the place where you can set the value of the *ProcessorExitStatus*. This attribute is particularly relevant when several processors are executed in a daisy-chained manner. See more about this *here* and *here*.

Specifically for processors with a looping workflow, these methods need to be subclassed.

The *get_items()* method.

> This is where the input collection of items is generated. Very likely this can be a list of files resulting from a glob or the rows of an input table or similar. This is required for all for loop type workflows (**for_loop**, **parallel_for_loop** and **parallel_for_loop_with_queue**) and must return an iterable object.

The *while_condition()* method.

> This is the point at which the decision is made to either continue or terminate the loop. This is required only for **while_loop** processors and must return a boolean value.

The *accept_item()* and *skip_item()* methods (optional).

> The execution of the for loop can be slightly modified using the *LoopingStatus* (see this *example*). If the current iteration was successful, then the user can decide to perform some actions, otherwise if the current iteration was failing, then some other actions can be taken.

The *consumer_start()*, *consumer_process()*, and *consumer_finish()* methods (optional).

> These hooks are specific to the **parallel_for_loop_with_queue** workflow and allow customization of the consumer thread that serializes post-processing. By default the consumer process delegates to *accept_item()* and *skip_item()*.

The *format_progress_message()* (optional).

> During the process execution, your console may look frozen because your CPU is working out your analysis, thus it may be relevant to have every now and then an update on the progress. The *Processor* will automatically display regular messages via the logging system about the progress (more or less every 10% of the total number of items), but the message it is using is rather generic and does not contain any information about the current item.
>
> By overloading this method, you can include information about the current item and customize the content of the message. You can use *item* to refer to the current item being processed. Here below is an example:

```
def format_progress_message(self):
    self.progress_message = f'{self.name} is processing {self.item}'
```

> Be aware that if your items are objects without a good __repr__ or string conversion, the output may be a little messy.

The *validate_configuration()* (optional).

> If your processor has several parameters, it might be a good idea that the configuration provided by the user via *processor.ActiveParameter* is validated. For this purpose the user can subclass the *validate_configuration()* method, where the validation of the configuration should silently take

place. If the configuration is invalid, then the method should raise a *InvalidConfigurationError*. You will read more on processor parameters and how to use it in one of the next *section*.

3.2.2 Customize and document you processor

You are a scientist, we know, the only text you like to write is the manuscript with your last research results. Nevertheless, documenting your code, in particular your processor is a very helpful approach. We strongly recommend to use *docstring* to give a description of what a processor does. If you do so, you will get a surprise when you will *generate your first steering file*.

It is also a very good practice to provide help/doc information to your parameters using the help_doc argument of the *ActiveParameter*. If you do so, your *first steering file* will be much more readable.

One more point, each *Processor* has a class attribute named *description*, this is a short string that is used by some user interfaces (like *RichInterface*) to make the progress bars more meaningful.

3.3 Processor parameters

One super power of MAFw is its capability to re-use code, that means less work, less bugs and more efficiency.

In order to boost code re-usability, one should implement Processor accomplishing one single task and possibly doing it with a lot of general freedom. If you have a processor to calculate the Threshold of B/W images and you have hard coded the threshold algorithm and tomorrow you decide to give a try to another algorithm, then you have to recode a processor that actually already exists.

The solution is to have processor parameters, kind of variables that can be changed in order to make your processor more general and more useful. A note of caution, if you opt for too many parameters, then it may become too difficult to configure your processor. As always, the optimal solution often lies in finding a balance.

You can have parameters implemented in the processor subclass as normal attributes, but then you would need to modify the code in order to change them and this is far from being practical. You can have them as specific variables passed to the processor init, but then you would need to code the command line script to pass this value or to implement a way to read a configuration file.

MAFw has already done all this for you as long as you use the right way to declare processor parameters.

Let us start with a bit of code.

```
class MyProcessor(Processor):
    """
    This is my wonderful processor.

    It needs to know the folder where the input files are stored.
    Let us put this in a processor parameter.

    :param input_folder: The input folder.
    """
    input_folder = ActiveParameter(name='input_folder', default=Path.cwd(),
                                   help_doc='This is the input folder for the file')

    def __init__(self, *args, **kwargs):
        super().__init(*args, **kwargs)

        # change the input folder to something else
        self.input_folder = Path(r'D:\data')

        # get the value of the parameter
        print(self.input_folder)
```

Some important notes: input_folder is defined as a class attribute rather then as an instance attribute, it is to say it is outside the init method. **This must always be the case for processor parameters**. The fact that the variable and

the parameter names are the same is not compulsory, but why do you want to make your life more complicated!

We have declared *input_folder* as an *ActiveParameter*, but if you put a break point in the code execution and you inspect the type of *self.input_folder* you will see that it is a Path object. This is because *ActiveParameter* is a descriptor, a very pythonic way to have a double interface for one single variable.

If your processor is subclassing the *__init__* method, then never forget to include right at the beginning a call to its super, this is ensuring that the processor, and its parameters, are properly initialized and ready to be used.

3.3.1 What's behind an ActiveParameter

> **Note**
>
> This is a very technical paragraph, you can skip it if you are not interested in the *behind the scene* of processor parameters.

Have a look at the code in the previous snippet. When you define *input_folder*, you use up to four parameters in the *ActiveParameter* init, but if you try to access those attributes from input_folder you will get an error message, because class Path does not have the attributes you are looking for. Where are then those values gone?

When you create an *ActiveParameter*, there is a lot of work behind the scene: each processor instance has a dictionary (*processor.Processor._processor_parameters*) where the parameter name is used as a key and the corresponding value is an instance of *processor.PassiveParameter*. This helper class has a very similar structure as the active counter part and it is used to store the current value of the parameter and all its metadata (default value and help string) defined when the active parameter was created.

The set and get dunder methods of the input_file are overloaded to operate directly on the private interface, more or less in the same way like when you define *property* setter and getter, but here it is done all automatically.

If you want to access the private interface, you can still do it. You can always use the processor *get_parameter()* using the parameter name as a key, to retrieve the *processor.PassiveParameter* instance. Theoretically you can even set the value of the parameter using the private interface (*set_parameter_value()*), but this is equivalent to set it via the public interface directly.

Let us summarize all this with an example:

```
from mafw.processor import Processor, ActiveParameter

class MyFavProcessor(Processor):
    useful_param = ActiveParameter('useful_param', default=0, help_doc='Important␣
↪parameter')

# create your processor, initialize the parameter with a keyword argument.
my_fav = MyFavProcessor(useful_param=10)

# print the value of useful_param in all possible ways
print(my_fav.useful_param)
print(my_fav.get_parameter('useful_param').value)

# change the value of useful_param in all possible ways
my_fav.useful_param += 1
my_fav.get_parameter('useful_param').value += 1

print(my_fav.useful_param)

# access other fields of the parameter
print(my_fav.get_parameter('useful_param').doc)
```

This is the output that will be generated:

```
10
10
12
Important parameter
```

3.3.2 Parameter configuration

We have seen how to add flexibility to a processor including parameters, but how do you configure the parameters?

You have probably noticed that *ActiveParameter* has the possibility to pass a default value, that is a very good practice, especially for very advanced parameters that will remain untouched most of the time.

If you want to set a value for a parameter, the easiest way is via the processor *__init__* method. The basic Processor accepts any number of keyword arguments that can be used exactly for this purpose. Just add a keyword argument named after the parameter and the processor will take care of the rest.

Have a look at the example below covering both the case of Active and Passive parameters:

Listing 3.1: Parameter setting via kwargs

```
class MyProcessor(Processor):
    active_param = ActiveParameter('value', default=0, help_doc='An active parameter')
    active_string = ActiveParameter('string', default=0, help_doc='An active parameter
↪')

my_p = MyProcessor(value=100, string='a better string', looper='single')

print(my_p.active_param)  # we get 100
assert my_p.active_param == 100

print(my_p.active_string)  # we get 'a better string'
assert my_p.active_string == 'a better string'
```

Note that the best way is to avoid to explicitly include the parameter names in the init signature. They will be collected anyhow through keyword arguments and registered automatically.

The second approach is to use a configuration object, it is to say a dictionary containing all the parameters key and value pairs. This is particularly handy when using a configuration file. Exactly for this reason, the configuration object can have one of the two following structures. In both cases the configuration object has to be passed to the class using the keyword *config*

Listing 3.2: Parameter setting via configuration object

```
@single_loop
class ConfigProcessor(Processor):
    p1 = ActiveParameter('param1', default='value')
    p2 = ActiveParameter('param2', default='value')


cp = ConfigProcessor(config=dict(param1='new_value', param2='better_value', param3=
↪'do not exists'))
assert cp.p1 == 'new_value'
assert cp.p2 == 'better_value'
dumped_config = cp.dump_parameter_configuration(option=2)
assert dumped_config == dict(param1='new_value', param2='better_value')

config = {'ConfigProcessor': {'param1': '1st', 'param2': '2nd'}}
cp = ConfigProcessor(config=config)
assert cp.p1 == '1st'
```

(continues on next page)

(continued from previous page)

```
assert cp.p2 == '2nd'
dumped_config = cp.dump_parameter_configuration()
assert config == dumped_config
```

Note

Migration from MAFw v1 to v2

If you were using MAFw already in its early day of version 1, you might remember that users could also use *processor.PassiveParameter* directly as processor parameters. They had the disadvantage of requiring the dot notation (*.value*) to access the data member, but everything else was identical to an *processor.ActiveParameter*. This was a consequence of the implementation of the old parameter initialisation. This procedure has been totally reviewed in version 2 and now there is no need anymore to use *processor.PassiveParameter*. On the contrary, the user is recommended not to use *processor.PassiveParameter*, but only their active counterpart.

3.3.3 Static parameter schema

When tooling such as GUI builders or template generators needs access to processor parameter metadata, the *parameter_schema()* classmethod exposes a serialisable list of *ParameterSchema* objects. Each schema entry includes the parameter name, resolved annotation, default, help string and booleans that flag list/dict expectations.

Because the schema derives solely from the *ActiveParameter* descriptors it does not instantiate any processor, enabling safe inspection of available parameters. The steering-file generator *generate_steering_file()* already consumes this API to build templates without creating processor objects.

```
from mafw.processor import Processor, ActiveParameter
from mafw.models.parameter_schema import ParameterSchema

class MyProcessor(Processor):
    option = ActiveParameter('option', default=1, help_doc='Set the option for tests')

schema: list[ParameterSchema] = MyProcessor.parameter_schema()
for entry in schema:
    print(entry.name, entry.default, entry.help)
```

The schema list always mirrors the order in which the *ActiveParameter* instances were defined (parents first, followed by subclass additions), and overrides replace the inherited entry without changing its relative position. This deterministic registry keeps configuration generators, steering templates, and GUIs in sync with the developer-defined order.

3.3.4 Saving parameter configuration

We have seen that we can configure processor using a dictionary with parameter / value pairs. This is very handy, because we can load toml file with the configuration to be used for all the processors we want to execute.

We don't want you to write toml file by hand, for this we have a function *dump_processor_parameters_to_toml()* that will generate an output file with all the parameter values.

But which value is stored? The default or the actual one? Good question! The actual answer is: *it depends*!

Let us have a look at the following example.

Listing 3.3: Processors definition

```
from mafw.processor import ActiveParameter, PassiveParameter

class ActiveParameterProcessor(Processor):
```

(continues on next page)

(continued from previous page)

```
    """A processor with one active parameter."""

    active_param = ActiveParameter('active_param', default=-1,
                                   help_doc='An active parameter with default value -1
↪')


class AnotherActiveParameterProcessor(Processor):
    """Another processor with one active parameter."""

    active_param = ActiveParameter('active_param', default=-1,
                                   help_doc='An active parameter with default value -1
↪')
```

We have defined two different processors with one parameter each.

Now let us have a look at what happens when we attempt to dump the configuration of these processors.

Listing 3.4: Case 1: all processor types

```
processor_list = [ActiveParameterProcessor, AnotherActiveParameterProcessor]

dump_processor_parameters_to_toml(processor_list, 'test1.toml')
```

We have not created any instances of the two processors, and the processor_list consists of two classes. When passed to the *dump_processor_parameters_to_toml()* function, no processors are instantiated. The helper reads each class's *parameter_schema()*, so it can fetch the declared default, the help text and other metadata without executing the constructor.

Because the schema copies the defaults from the original *ActiveParameter*, the dumped value for every parameter is the declared default when no initial value is provided in the descriptor.

The output of the *first example* will show the two processors with their parameters listed and connected to their default values.

Listing 3.5: Output of case 1: test1.toml

```
[ActiveParameterProcessor] # A processor with one active parameter.
active_param = -1 # An active parameter with default value -1

[AnotherActiveParameterProcessor] # Another processor with one active parameter.
active_param = -1 # An active parameter with default value -1
```

Now, let us take a step forward and add a bit more complexity to the situation. See the next snippet:

Listing 3.6: Case 2: mixed instances and classes

```
# create an instance of ActiveParameterProcessor with a specific value of the␣
↪parameter
# but we will include the class in the processor list.
active_processor_instance = ActiveParameterProcessor(active_param=100)

# create an instance of AnotherActiveParameterProcessor with a specific value of the␣
↪parameter
# and we will submit the instance
another_active_processor_instance = AnotherActiveParameterProcessor(active_param=101)


processor_list = [
```

(continues on next page)

(continued from previous page)

```
    ActiveParameterProcessor,  # a class
    another_active_processor_instance,  # an instance
]
dump_processor_parameters_to_toml(processor_list, 'test2.toml')
```

This time we have a mixed list, the first item is still a class, so the helper will reuse the static schema to obtain the default -1 and no constructor is invoked. For the second entry we are passing an actual instance, so the dumped value reflects the runtime value 101 set when the instance was created.

Here is the produced configuration file:

Listing 3.7: Output of case 2: test2.toml

```
[ActiveParameterProcessor] # A processor with one active parameter.
active_param = -1 # An active parameter with default value -1

[AnotherActiveParameterProcessor] # Another processor with one active parameter.
active_param = 101 # An active parameter with default value -1
```

Ultimately, the values that will be output depends on the fact if you pass an instance of the processor or its class. But in the end, this is not really important because the user should dump TOML configuration files as a sort of template to be checked, modified and adapted by the user for a specific run.

3.3.5 Parameter and processor inheritance

Parameters, like any other class attributes and methods, can be inherited via subclassing and this can turn out to be very useful. Imagine to have a full class of processors having some parameters in common (the most simple example is the base path for the produced output), then the user might consider to have an intermediate subclass between MAFw Processor and their final Processors defining all these common parameters.

Have a look at the example below:

```
@single_loop
class BaseProcessor(Processor):
    my_param = ActiveParameter('my_param', default=5, help_doc='An integer value')

class SameDefaultProcessor(BaseProcessor):
    pass

class OtherDefaultProcessor(BaseProcessor):
    new_defaults = {'my_param': 885}

dump_processor_parameters_to_toml([BaseProcessor, SameDefaultProcessor,␣
↪OtherDefaultProcessor], 'test.toml')
```

All processors have a parameter names *my_param*. Since the parameter has been defined only once (in the base processor), it is born with the same default value for all subclasses. But this can be changed setting the processor instance variable *new_defaults* as done in the OtherDefaultProcessor. In fact, the TOML configuration file is exactly showing this situation:

```
[BaseProcessor]
my_param = 5 # An integer value

[SameDefaultProcessor]
my_param = 5 # An integer value

[OtherDefaultProcessor]
my_param = 885 # An integer value
```

3.3.6 Parameter names

The name of a parameter must be a valid Python identifier and there are a bunch of names that are reserved for internal use. The validity of a parameter name is automatically checked, so you will not be able to mess up the system.

3.3.7 Parameter typing

When creating an Active parameter, you have the option to directly specify the parameter's type using the typing template, but you can also do it, and probably in a simpler way assigning a reasonable default value. While this is not really important for numbers, it is extremely important if you want to interpret string as Path object.

If you declare the default value as a Path, for example Path.cwd(), then the string read from the configuration file will be automatically converted in a Path.

> **Note**
>
> If you intend to have a float parameter, use a decimal number, for example 0., as default, otherwise the interpreter will assume it is an integer and convert to int the parameter being read from the configuration file.

One more note about parameters. Theoretically speaking you could also have custom objects / classes as parameters, but this will become a problem when you will be loading the parameters from a TOML file. Actually two problems:

1. The TOML writer is not necessarily able to convert your custom type to a valid TOML type (number, string...). If your custom type has a relatively easy string representation then you can add an encoder to the TOML writer and teach it how to write your object. See for example the encoder for the Path object.

```
def path_encoder(obj: Any) -> Item:
    """Encoder for PathItem."""
    if isinstance(obj, PosixPath):
        return PathItem.from_raw(str(obj), type_=StringType.SLB, escape=False)
    elif isinstance(obj, WindowsPath):
        return PathItem.from_raw(str(obj), type_=StringType.SLL, escape=False)
    else:
        raise ConvertError
```

2. Even though you managed to write your class to the TOML steering file, you have now the problem of reading back the steering file information and build your custom type with that.

One way to overcome this limitation might be to write to the steering file the `__repr__` of your custom class and at read back time to use eval to transform it back to your class. This below would be a more concrete implementation:

python

```
import tomlkit
from tomlkit.items import  String, Item, ConvertError
from tomlkit.toml_file import TOMLFile

class MyClass:
    """A custom class to be used as a processor parameter"""
    def __init__(self, a, b):
        self.a = a
        self.b = b

    def __repr__(self):
        """IT MUST BE IMPLEMENTED"""
```

(continues on next page)

(continued from previous page)

```
        return f"{self.__class__.__name__}({self.a}, {self.b})"

class MyClassItem(String):
    """TOML item representing a MyClass"""
    def unwrap(self) -> MyClass:
        return MyClass(*super().unwrap())

def my_class_encoder(obj: MyClassItem) -> Item:
    """Encoder for MyClassItem."""
    if isinstance(obj, MyClass):
        # we write the class as a string using the class repr.
        return MyClassItem.from_raw(repr(obj))
    else:
        raise ConvertError

# register the encoder
tomlkit.register_encoder(my_class_encoder)

# ------------------
# write to TOML file
# ------------------

my_class = MyClass(10,24)
doc = tomlkit.document()
doc.add('my_class', my_class)
doc.add('simple_i', 15)

with open('test.toml', 'w') as fd:
    tomlkit.dump(doc, fd)

# ------------------------
# read back from TOML file
# ------------------------

doc = TOMLFile('test.toml').read()
read_back_class = eval(doc['my_class'])
try:
    simple_i = eval(doc['simple_i'])
except TypeError:
    simple_i = doc['simple_i']

assert isinstance(read_back_class, MyClass)
assert read_back_class.a == my_class.a

assert isinstance(simple_i, int)
assert simple_i == 15
```

TOML

```
my_class = "MyClass(10, 24)"
simple_i = 15
```

This approach, even though possible, is rather risky, we all know how dangerous `eval` can be especially when using it directly with information coming from external files. Furthermore, it is worth considering whether having a custom class as a parameter in a processor is truly necessary. Often, there are simpler and safer alternatives available.

3.4 What's next

You reached the end of the first part. It means that by now you have understood what a processor is and how you can subclass the basic class to implement your analytical tasks, both with looping or single shot workflow. You learned a lot about parameters and how you can configure it.

The next section will be about chaining more processors one after the other using the *ProcessorList*.

CHAPTER 4

PROCESSOR EXAMPLES

From this page, you can see a few example of processors in order to simplify the creation of your first processor sub class.

4.1 Simple and looping processors

The first two examples of this library are demonstrating how you can implement a simple processor that execute all calculations in one go and a looping processor where you need to loop over a list of items to get either a cumulative results.

AccumulatorProcessor is calculating the sum of the first N integer numbers in a loop. The processor takes the *last_number* as an input to include and put the output in the *accumulated_value* parameter. This process is very inefficient, but it is here to demonstrate how to subclass a looping processor.

```
class AccumulatorProcessor(Processor):
    r"""
    A processor to calculate the sum of the first n values via a looping approach.

    In mathematical terms, this processor solves this easy equation:

    .. math::

        N = \sum_{i=0}^{n}{i}

    by looping. It is a terribly inefficient approach, but it works as a demonstration of the looping structure.

    The user can get the results by retrieving the `accumulated_value` parameter at the end of the processor
    execution.
    """

    last_value = ActiveParameter('last_value', default=100, help_doc='Last value of the series')

    def __init__(self, *args, **kwargs):
        """Constructor parameters:

        :param last_value: The `n` in the equation above. Defaults to 100
        :type last_value: int
```

(continues on next page)

(continued from previous page)

```
        :param accumulated_value: The `N` in the equation above at the end of the process.
        :type accumulated_value: int
        """
        super().__init__(*args, **kwargs)
        self.accumulated_value: int = 0

    def start(self):
        """Resets the accumulated value to 0 before starting."""
        super().start()
        self.accumulated_value = 0

    def get_items(self) -> list[int]:
        """Returns the list of the first `last_value` integers."""
        return list(range(self.last_value))

    def process(self):
        """Increase the accumulated value by the current item."""
        self.accumulated_value += self.item
```

GaussAdder is calculating exactly the same result using the Gauss formula, eliminating the need for any looping. Indeed the looping is disabled and the output is the same.

```
class GaussAdder(Processor):
    r"""
    A processor to calculate the sum of the first n values via the so called *Gauss formula*.

    In mathematical terms, this processor solves this easy equation:

    .. math::

        N = \frac{n * (n - 1)}{2}

    without any looping

    The user can get the results by retrieving the `sum_value` parameter at the end of the processor
    execution.
    """

    last_value = ActiveParameter('last_value', default=100, help_doc='Last value of the series.')

    def __init__(self, *args, **kwargs):
        """
        Constructor parameters:

        :param last_value: The `n` in the equation above. Defaults to 100
        :type last_value: int
        :param sum_value: The `N` in the equation above.
        :type sum_value: int
        """
        super().__init__(looper=LoopType.SingleLoop, *args, **kwargs)
        self.sum_value: int = 0

```

(continues on next page)

(continued from previous page)

```
31      def start(self):
32          """Sets the sum value to 0."""
33          super().start()
34          self.sum_value = 0
35
36      def process(self):
37          """Compute the sum using the Gauss formula."""
38          self.sum_value = int(self.last_value * (self.last_value - 1) / 2)
```

If you carefully look at line 28, you will notice that in the GaussAdder constructor, the looper option is set to SingleLoop and as we have *seen*, it means that that the processor will follow the single loop execution workflow.

The definition of the looper parameter in the init method can be sometimes hard to remember and unpractical especially if you have to overload the init method just to set the value of the looper. In such circumstances the use of a class decorator can be very handy. MAFw makes you available five class decorators for this purpose, to transform a processor in a *single loop*, a *for loop*, a *parallel for loop*, a *parallel for loop with queue* or a *while loop*.

Using the decorator approach the GaussAdder above can be re-written in this way:

```
@single_loop
class GaussAdder(Processor):
    # the rest of the implementation remains the same
```

And here below is an example of execution of the two.

```
from mafw.examples.sum_processor import GaussAdder, AccumulatorProcessor

n = 35

# create the two processors
accumulator = AccumulatorProcessor(last_value=n)
gauss = GaussAdder(last_value=n)

# execute them
accumulator.execute()
gauss.execute()

# print the calculated results
print(accumulator.accumulated_value)
print(gauss.sum_value)
```

This will generate the following output:

```
595
595
```

4.2 Modify the *for loop* cycle using the `LoopingStatus`

In a looping processor, the *process()* method is invoked inside a loop, but the user can decide to skip a certain item and even to interrupt the (abort or quit) the loop.

The tool to achieve this is the *LoopingStatus*. This is set to Continue at the beginning of each iteration, but the user can turn it Skip, Abort or Quit inside the implementation of *process()*.

When set to Skip, a special callback is invoked *skip_item()* where the user can do actions accordingly. When set to Abort or Quit, the loop is broken and the user can decide what to do in the *finish()* method. Those two statuses seams to be redundant, but this gives the user the freedom to decide if everything was wasted (Abort) or if what done so far was still acceptable (Quit).

Here below is the implementation of a simple processor demonstrating such a functionality.

```python
class ModifyLoopProcessor(Processor):
    """
    Example processor demonstrating how it is possible to change the looping structure.

    It is a looping processor where some events will be skipped, and at some point one event will trigger an abort.
    """

    total_item: ActiveParameter[int] = ActiveParameter('total_item', default=100, help_doc='Total item in the loop.')
    items_to_skip: ActiveParameter[list[int]] = ActiveParameter(
        'items_to_skip', default=[12, 16, 25], help_doc='List of items to be skipped.'
    )
    item_to_abort: ActiveParameter[int] = ActiveParameter('item_to_abort', default=65, help_doc='Item to abort')

    def __init__(self, *args, **kwargs):
        """
        Processor Parameters:

        :param total_item: The total number of items
        :type total_item: int
        :param items_to_skip: A list of items to skip.
        :type items_to_skip: list[int]
        :param item_to_abort: The item where to trigger an abort.
        :type item_to_abort: int

        """
        super().__init__(*args, **kwargs)
        self.skipped_items: [list[int]] = []
        """A list with the skipped items."""

    def start(self):
        """Resets the skipped item container."""
        super().start()
        self.skipped_items = []

    def get_items(self) -> range:
        """Returns the list of items, the range from 0 to total_item."""
        return range(self.total_item)

    def process(self):
        """Processes the item"""
        if self.item in self.items_to_skip:
            self.looping_status = LoopingStatus.Skip
            return
        if self.item == self.item_to_abort:
            self.looping_status = LoopingStatus.Abort
            return

    def skip_item(self):
        """Add skipped item to the skipped item list."""
        self.skipped_items.append(self.item)
```

And here below is how the processor can be used.

```python
import random
from mafw.examples.loop_modifier import ModifyLoopProcessor

# generate a random number corresponding to the last item
last_value = random.randint(10, 1000)

# get a sample with event to be skipped
skip_items = random.sample(range(last_value), k=4)

# find an event to abort after the last skipped one
max_skip = max(skip_items)
if max_skip + 1 < last_value:
    abort_item = max_skip + 1
else:
    abort_item = last_value - 1

# create the processor and execute it
mlp = ModifyLoopProcessor(total_item=last_value, items_to_skip=skip_items, item_to_
↪abort=abort_item)
mlp.execute()

# compare the recorded skipped items with the list we provided.
assert mlp.skipped_items == list(sorted(skip_items))

# check that the last item was the abort item.
assert mlp.item == abort_item
```

4.3 For and while loop execution workflow

We have seen in the previous chapter that there are different type of loopers and in the previous section we have seen in practice the execution workflow of a single loop and a while loop processor.

In this example, we will explore the difference between the **for loop** and the **while loop** execution workflow. Both processors will run the *processor.Processor.process()* method inside a loop, but for the former we will loop over a pre-established list of items, while for the latter we will continue repeating the process until a certain condition is valid.

Both processors will work with prime number and we will use this *helper function* to check if an integer number is prime or not.

```python
def is_prime(n: int) -> bool:
    """
    Check if n is a prime number.

    :param n: The integer number to be checked.
    :type n: int
    :return: True if n is a prime number. False, otherwise.
    :rtype: bool
    """
    prime = True
    if n < 2:
        prime = False
    elif n == 2:
        prime = True
    elif n % 2 == 0:
        prime = False
    else:
```

(continues on next page)

(continued from previous page)

```
        sqrt_n = int(math.floor(math.sqrt(n)))
        for i in range(3, sqrt_n + 1, 2):
            if n % i == 0:
                prime = False

    return prime
```

The task of the **for loop** processor is to find all prime numbers included in a given user defined range of integer numbers. In other words, we want to find all prime numbers between 1000 and 2000, for example. The brute force approach is to start a loop on 1000, check if it is prime and if not check the next one until you get to 2000. If a number is actually prime, then store it in a list for further use.

For the sake of clarity, along with the *API documentation*, we are copying here also the processor source code.

```
@for_loop
class FindPrimeNumberInRange(Processor):
    """
    An example processor to find prime numbers in the defined interval from ``start_
↪from`` to ``stop_at``.

    This processor is meant to demonstrate the use of a for_loop execution workflow.

    Let us say we want to select only the prime numbers in a user defined range. One␣
↪possible brute force approach is
    to generate the list of integers between the range extremes and check if it is␣
↪prime or not. If yes,
    then add it to the list of prime numbers, if not continue with the next element.

    This is a perfect application for a loop execution workflow.
    """

    start_from = ActiveParameter('start_from', default=50, help_doc='From which␣
↪number to start the search')
    stop_at = ActiveParameter('stop_at', default=100, help_doc='At which number to␣
↪stop the search')

    def __init__(self, *args: Any, **kwargs: Any):
        """
        Processor parameters:

        :param start_from: First element of the range under investigation.
        :type start_from: int
        :param stop_at: Last element of the range under investigation.
        :type stop_at: int
        """
        super().__init__(*args, **kwargs)
        self.prime_num_found: list[int] = []
        """The list with the found prime numbers"""
```

This is the class definition with its constructor. As you can see, we have decorated the class with the *for loop decorator* even though it is not strictly required because the for loop is the default execution workflow.

We have added two processor parameters, the `start_from` and the `stop_at` to allow the user to specify a range on interest where to look for prime numbers.

In the init method, we create a list of integer to store all the prime numbers that we will finding during the process.

Now let us overload all compulsory methods for a **for loop** processor.

```python
def get_items(self) -> Collection[Any]:
    """
    Overload of the get_items method.

    This method must be overloaded when you select a for loop workflow.

    Here we generate the list of odd numbers between the start and stop that we need to check.
    We also check that the stop is actually larger than the start, otherwise we print an error message, and we
    return an empty list of items.

    :return: A list of odd integer numbers between start_from and stop_at.
    :rtype: range[int]
    """
    if self.start_from >= self.stop_at:
        log.critical(f'{self.start_from} must be smaller than {self.stop_at}')
        return []

    if self.start_from != 2 and self.start_from % 2 == 0:
        self.start_from += 1

    if self.stop_at != 2 and self.stop_at % 2 == 0:
        self.stop_at -= 1

    return range(self.start_from, self.stop_at, 2)
```

The get items method is expected to return a range, that will be processed by the *processor.Processor.process()* method. It is absolutely compulsory to overload this method, otherwise the whole loop structure will not have a list to loop over.

And now, let us have a look at the three stages: start, process and finish.

```python
def start(self) -> None:
    """
    Overload of the start method.

    **Remember:** to call the super method when you overload the start.

    In this specific case, we just make sure that the list of found prime numbers is empty.
    """
    super().start()
    self.prime_num_found = []

def process(self) -> None:
    """
    The process method.

    In this case, it is very simple. We check if :attr:`.processor.Processor.item` is a prime number, if so we added to the list,
    otherwise we let the loop continue.
    """
    if is_prime(self.item):
        self.prime_num_found.append(self.item)

def finish(self) -> None:
```

(continues on next page)

(continued from previous page)

```
    """
    Overload of the finish method.

    **Remember:** to call the super method when you overload the finish method.

    In this case, we just print out some information about the prime number found in the range.
    """
    super().finish()
    log.info(
        'Found %s prime numbers in the range from %s to %s'
        % (len(self.prime_num_found), self.start_from, self.stop_at)
    )
    if len(self.prime_num_found):
        log.info('The smallest is %s', self.prime_num_found[0])
        log.info('The largest is %s', self.prime_num_found[-1])
```

These three methods are the core of the execution workflow, so it is obvious that you have to overload them. Keep in mind to always include a call to the super method when you overload the start and finish because they perform some tasks also in the basic processor implementation. The code is written in a straightforward manner and includes clear, thorough explanations in the docstring.

The looping parameters: *processor.Processor.i_item*, *processor.Processor.n_item* and *processor.Processor.item* can be used while implementing the *process()* and *finish()*. The *n_item* is calculated soon after the collection of items is returned, while *item*, *i_item* are assigned in the for loop as the current item and its enumeration.

Optionally, one can overload the *format_progress_message()* in order to generate a nice progress message informing the user that something is happening. This is an example:

```
def format_progress_message(self) -> None:
    self.progress_message = (
        f'Checking integer number: {self.item}, already found {len(self.prime_num_found)} prime numbers'
    )
```

The task for the **while loop** processor is again about prime number finding but different. We want to find a certain number of prime numbers starting from an initial value. We cannot generate a list of integer number and loop over that in the *FindPrimeNumberInRange*, but we need to reorganize our workflow in order to loop until the number of found primes is equal to the requested one.

This is how such a task can be implemented using the while loop execution framework. You can find the example in the *API documentation* and an explanation here below.

Let us start again from the class definition.

```
@while_loop
class FindNPrimeNumber(Processor):
    """
    An example of Processor to search for N prime numbers starting from a given starting integer.

    This processor is meant to demonstrate the use of a while_loop execution workflow.

    Let us say we need to find 1000 prime numbers starting from 12347. One possible brute force approach to solve this
    problem is to start checking if the initial value is a prime number. If this is not the case, then check the next
    odd number. If it is the case, then add the current number to the list of found
```

(continues on next page)

(continued from previous page)

```
    prime numbers and continue until
        the size of this list is 1000.

        This is a perfect application for a while loop execution workflow.
        """

        prime_num_to_find = ActiveParameter(
            'prime_num_to_find', default=100, help_doc='How many prime number we have to find'
        )
        start_from = ActiveParameter('start_from', default=50, help_doc='From which number to start the search')

        def __init__(self, *args: Any, **kwargs: Any):
            """
            Processor parameters:

            :param prime_num_to_find: The number of prime numbers to be found.
            :type prime_num_to_find: int
            :param start_from: The initial integer number from where to start the search.
            :type start_from: int
            """
            super().__init__(*args, **kwargs)
            self.prime_num_found: list[int] = []
            """The list with the found prime numbers"""
```

The first difference compared to the previous case is the use of the *while_loop()* decorator, this time it is really necessary to specify the processor *LoopType* because the while loop is not the default strategy.

The processor has two parameters, the number of prime number to find and from where to start. Similarly as before, in the init method, we define a list of integer to store all the prime numbers that we have found.

For while loop processor, we don't have a list of items, but we need to have a condition either to continue or to stop the loop. For this reason we need to overload the *while_condition()* method, keeping in mind that we return True if we want the cycle to continue for another iteration and False otherwise.

Here is the implementation of the *while_condition()* for the *FindNPrimeNumber*.

```
def while_condition(self) -> bool:
    """
    Define the while condition.

    First, it checks if the prime_num_to_find is positive. Otherwise, it does not make sense to start.
    Then it will check if the length of the list with the already found prime numbers is enough. If so, then we can
    stop the loop return False, otherwise, it will return True and continue the loop.

    Differently from the for_loop execution, we are responsible to assign the value to the looping variables
    :attr:`.processor.Processor.i_item`, :attr:`.processor.Processor.item` and :attr:`.processor.Processor.n_item`.

    In this case, we will use the :attr:`.processor.Processor.i_item` to count how many prime numbers we have found and :attr:`.processor.Processor.n_item`
    will be our target. In this way, the progress bar will work as expected.

    In the while condition, we set the :attr:`.processor.Processor.i_item` to the
```

(continues on next page)

(continued from previous page)

```
↪current length of the found prime number list.

    :return: True if the loop has to continue, False otherwise
    """
    if self.prime_num_to_find <= 0:
        log.warning('You requested to find a negative number of prime numbers. It␣
↪makes no sense.')
        return False

    self.i_item = len(self.prime_num_found)
    return self.i_item < self.prime_num_to_find
```

For a while loop, it is not easy to define an enumeration parameter and also the total number of items might be misleading. It is left to the user to decide if they want to use them or not. If yes, their definition and incrementation is under their responsibility. For this processor, it was natural to consider the requested number of primes as the *processor.Processor.n_item* and consequently the value of *processor.Processor.i_item* can be utilized to keep track of the quantity of prime numbers that have already been discovered. This choice is very convenient because then progress bar that uses *processor.Processor.i_item* and *processor.Processor.n_item* to calculate the progress will show the actual progress. In case, you do not have any way to assign a value to *processor.Processor.n_item*, do not do it, or set it to None. In this way, the progress bar will display an indeterminate progress . You can set the value of *processor.Processor.n_item* either in the *start()* or in the *while_condition()*, with a performance preference with the first option because it is executed only once before the start of the loop.

Here below is the implementation of the three stages.

```
def start(self) -> None:
    """
    The overload of the start method.

    **Remember:** The start method is called just before the while loop is started.␣
↪So all instructions in this
    method will be executed only once at the beginning of the process execution.␣
↪Always put a call to its `super`
    when you overload start.

    First, we empty the list of found prime numbers. It should not be necessary, but␣
↪it makes the code more readable.
    Then set the :attr:`.processor.Processor.n_item` to the total number of prime␣
↪numbers we need to find. In this way, the progress bar
    will display useful progress.

    If the start value is smaller than 2, then let's add 2 to the list of found prime␣
↪number and set our first
    item to check at 3. In principle, we could already add 3 as well, but maybe the␣
↪user wanted to find only 1
    prime number, and we are returning a list with two, that is not what he was␣
↪expecting.

    Since prime numbers different from 2 can only be odd, if the starting number is␣
↪even, increment it already by
    1 unit.
    """
    super().start()
    self.prime_num_found = []
    self.n_item = self.prime_num_to_find
    if self.start_from < 2:
```

(continues on next page)

(continued from previous page)

```
        self.prime_num_found.append(2)
        self.start_from = 3

    if self.start_from % 2 == 0:
        self.item = self.start_from + 1
    else:
        self.item = self.start_from

def process(self) -> None:
    """
    The overload of the process method.

    **Remember:** The process method is called inside the while loop. It has access to the looping parameters:
    :attr:`.processor.Processor.i_item`, :attr:`.processor.Processor.item` and :attr:`.processor.Processor.n_item`.

    In our specific case, the process contains another while loop. We start by checking if the current
    :attr:`.processor.Processor.item` is a prime number or not. If so, then we have found the next prime number, we add it to the list,
    we increment by two units the value of :attr:`.processor.Processor.item` and we leave the process method ready for the next iteration.

    If :attr:`.processor.Processor.item` is not prime, then increment it by 2 and check it again.
    """
    while not is_prime(self.item):
        self.item += 2
    self.prime_num_found.append(self.item)
    self.item += 2

def finish(self) -> None:
    """
    Overload of the finish method.

    **Remember:** The finish method is called only once just after the last loop interaction.
    Always put a call to its `super` when you overload finish.

    The loop is over, it means that the while condition was returning false, and now we can do something with our
    list of prime numbers.
    """
    super().finish()
    log.info('Found the requested %s prime numbers' % len(self.prime_num_found))
    log.info('The smallest is %s', self.prime_num_found[0])
    log.info('The largest is %s', self.prime_num_found[-1])
```

Let us have a look at the *FindNPrimeNumber.start()*. First of all we set the value of *processor.Processor.n_item* to our target value of primes. We use the *processor.Processor.item* to store the current integer number being tested, so we initialize it to start_from or the first not prime odd number following it. In the *FindNPrimeNumber.process()* we need to include another while loop, this time we need to check the current value of *processor.Processor.item* if it is a prime number. If yes, then we add it to the storage list, we increment it by two units (*remember that for while loop processors it is your responsibility to increment the loop parameters*) and we get ready for the next loop iteration. As for the other processor, we *FindNPrimeNumber.finish()* printing some statistics.

4.4 Parallel execution

4.4.1 Parallel for loop execution

Parallel for loops distribute items across a thread pool when Python runs in free-threading mode. The execution order is not guaranteed, so any shared state must be protected (for instance with a lock).

Here below is a processor variant that evaluates prime numbers in parallel and stores the results using a thread-safe list update.

```
@parallel_for_loop
class FindPrimeNumberInRangeParallel(Processor):
    """
    An example processor to find prime numbers in parallel within the defined interval.

    This processor demonstrates the use of the parallel for loop execution workflow and highlights the need for
    thread-safe handling of shared state.
    """

    start_from = ActiveParameter('start_from', default=50, help_doc='From which number to start the search')
    stop_at = ActiveParameter('stop_at', default=100, help_doc='At which number to stop the search')

    def __init__(self, *args: Any, **kwargs: Any):
        """
        Processor parameters:

        :param start_from: First element of the range under investigation.
        :type start_from: int
        :param stop_at: Last element of the range under investigation.
        :type stop_at: int
        """
        super().__init__(*args, **kwargs)
        self.prime_num_found: list[int] = []
        """The list with the found prime numbers"""
        self._lock = threading.Lock()
        """Lock to protect shared list updates in parallel execution."""

    def format_progress_message(self) -> None:
        self.progress_message = (
            f'Checking integer number: {self.item}, already found {len(self.prime_num_found)} prime numbers'
        )

    def get_items(self) -> Collection[Any]:
        """
        Return a list of odd integer numbers between start_from and stop_at.

        :return: A list of odd integer numbers between start_from and stop_at.
        :rtype: list[int]
        """
        if self.start_from >= self.stop_at:
            log.critical(f'{self.start_from} must be smaller than {self.stop_at}')
            return []

```

(continues on next page)

(continued from previous page)

```
44          if self.start_from != 2 and self.start_from % 2 == 0:
45              self.start_from += 1
46
47          if self.stop_at != 2 and self.stop_at % 2 == 0:
48              self.stop_at -= 1
49
50          return range(self.start_from, self.stop_at, 2)
51
52      def start(self) -> None:
53          """Reset the list of found prime numbers."""
54          super().start()
55          self.prime_num_found = []
56
57      def process(self) -> None:
58          """
59          Process a single item in parallel.
60
61          The update of shared results is guarded by a lock to remain thread-safe.
62          """
63          if is_prime(self.item):
64              with self._lock:
65                  self.prime_num_found.append(self.item)
66
67      def finish(self) -> None:
68          """Print summary information about the prime numbers found."""
69          super().finish()
70          log.info(
71              'Found %s prime numbers in the range from %s to %s'
72              % (len(self.prime_num_found), self.start_from, self.stop_at)
73          )
74          if len(self.prime_num_found):
75              log.info('The smallest is %s', min(self.prime_num_found))
76              log.info('The largest is %s', max(self.prime_num_found))
```

Looking at the code you might have noticed how similar the implementation is with respect to what was done for the standard serial for loop. There are only three tiny differences highlighted in the code. The first one is actually at the first line, where we use a decorator to force our processor to follow a parallel for loop workflow. The second difference is at line 25 where we define a lock inside the processor class and we use it at line 64 when we are appending the result of the calculation to the shared list of found prime numbers.

According to the python documentation, built-in objects like list are already multi-thread safe also in the free-threading version, but it is still recommended to protect them with a threading lock to avoid concurrent modifications.

Where is the threading pool executor, the worker definition and all the other usual suspects of a multi-threading application? Well hidden in the MAFw machinery and you do not need to use them, unless you want to see them in action. In such a case have a look at the implementation of *processor.Processor._process_parallel_for_loop()*.

4.4.2 Parallel for loop with consumer / producer pattern

In the previous *example* we have seen a very common approach to parallel processing where a pool of workers are processing all items in the collection. This approach is very effective when the serialization bottleneck (the lock protected append in the list) is not too heavy. The risk is that your workers are ready to deliver their result but they have to wait for the list to be unlocked.

An alternative approach is the *so-called* **consumer/producer** pattern as we have explained it in a *previous chapter*, here below you will be seeing the same processor implemented according to this architectural design.

```python
@parallel_for_loop_with_queue
class FindPrimeNumberInRangeParallelWithQueue(Processor):
    """
    An example processor to find prime numbers in parallel using a consumer queue.

    The processing work is distributed across multiple threads, while accepting results happens on a single
    consumer thread, avoiding the need for explicit locks on shared state.
    """

    start_from = ActiveParameter('start_from', default=50, help_doc='From which number to start the search')
    stop_at = ActiveParameter('stop_at', default=100, help_doc='At which number to stop the search')

    def __init__(self, *args: Any, **kwargs: Any):
        """
        Processor parameters:

        :param start_from: First element of the range under investigation.
        :type start_from: int
        :param stop_at: Last element of the range under investigation.
        :type stop_at: int
        """
        super().__init__(*args, **kwargs)
        self.prime_num_found: list[int] = []
        """The list with the found prime numbers"""

    def format_progress_message(self) -> None:
        self.progress_message = (
            f'Checking integer number: {self.item}, already found {len(self.prime_num_found)} prime numbers'
        )

    def get_items(self) -> Collection[Any]:
        """
        Return a range of odd integer numbers between start_from and stop_at.

        :return: A range of odd integer numbers between start_from and stop_at.
        :rtype: range[int]
        """
        if self.start_from >= self.stop_at:
            log.critical(f'{self.start_from} must be smaller than {self.stop_at}')
            return []

        if self.start_from != 2 and self.start_from % 2 == 0:
            self.start_from += 1

        if self.stop_at != 2 and self.stop_at % 2 == 0:
            self.stop_at -= 1

        return range(self.start_from, self.stop_at, 2)

    def start(self) -> None:
        """Reset the list of found prime numbers."""
        super().start()
        self.prime_num_found = []
```

(continues on next page)

(continued from previous page)

```
54
55      def process(self) -> int | None:
56          """
57          Process a single item in parallel.
58
59          The consumer thread will handle accepted results without additional locking.
60          """
61          if is_prime(self.item):
62              return self.item
63          self.looping_status = LoopingStatus.Skip
64          return None
65
66      def accept_item(self, result: LoopResult | None = None) -> None:
67          """Store the prime number on the consumer thread."""
68          if result is not None:
69              self.prime_num_found.append(result.payload)
70          else:
71              self.prime_num_found.append(self.item)
72
73      def finish(self) -> None:
74          """Print summary information about the prime numbers found."""
75          super().finish()
76          log.info(
77              'Found %s prime numbers in the range from %s to %s'
78              % (len(self.prime_num_found), self.start_from, self.stop_at)
79          )
80          if len(self.prime_num_found):
81              log.info('The smallest is %s', min(self.prime_num_found))
82              log.info('The largest is %s', max(self.prime_num_found))
```

The big difference here is conceptual. The *process* method is invoked by the pool of producers while the *accept_item* / *skip_item* methods are called by the consumer thread. So we need a way to connect these different threads using a queue. You can have a look at the implementation= of all this plumbing in *processor.Processor._process_parallel_for_loop_with_queue()*, but you do not have to, you can also just focus on the example at the line 62 and 64. Normally the process method is returning None because if we need to store the process results we normally use an instance attribute, but here we cannot do so, because the process and the accept_item are living in two different threads.

The strategy is to make process to return the result object, in this case a prime number, and we can use the LoopingStatus flag to decide what to do with the result. In the present case, we will keep all prime numbers and we will just dispose all the others.

The default implementation of the consumer worker is this one (but you can overload it they way you like it):

```
def consumer_process(self, loop_result: LoopResult | None = None) -> None:
    """
    Handle processed items in the consumer thread for the parallel queue loop.

    By default this dispatches to :meth:`accept_item` or :meth:`skip_item` based on␣
↪the looping status. The method
    can optionally accept a :class:`~mafw.models.loop_payloads.LoopResult` payload.
    """
    if loop_result is None:
        loop_result = self._build_loop_result(None, 0.0)
    if self.looping_status == LoopingStatus.Continue:
        self._call_with_optional_payload(self.accept_item, loop_result)
    elif self.looping_status == LoopingStatus.Skip:
        self._call_with_optional_payload(self.skip_item, loop_result)
```

If you look at the *accept_item* you can see that the method is receiving an optional `LoopResult` object. The loop_result is None if the processor is being executed in a single thread fallback mode, if not, it means that the processor is being executed in a multi-thread environment and the result of the process has been wrapped inside a LoopResult class, added to the shared queue and then retrieved by consumer thread.

The loop result contains several information but the most important is its payload that is exactly what process returned. If item is a prime number, then the payload will be the prime number, otherwise it will be None, but it will never reach the accept_item because the *consumer_process* will have dispatched it to the skip_item (that in this implementation is just a no-op).

For this architecture to be effective, the queue must remain relatively empty otherwise the operating system will spend all its time to try to find an empty slot in the queue. In order for this to happen, the consumer thread must last longer than consumer one. If processing a single item is too short, you can combine several items all together and instead of transferring one single output we will transfer a list of results. This technique, called **batching**, can improve the execution time by large.

4.4.3 Which implementation to prefer?

As always it depends. It mainly depends on how computational intense is the work that the workers have to accomplish, how slow is the serialization process (either in the consumer thread or by the lock) and how big is the overhead to start all the threads and to wrap all results.

Let's see a very practical example: run the three prime number finder processors on the same range from 5 to 99999. The numbers to be tested are many, but so small that every process takes less than 1 µs. In this case, the serial execution will be only slightly worse than the Parallel execution and even faster than the ParallelWithQueue.

If you change the range and you search prime numbers between 9999956789 and 9999997789 the situation is totally changed. The wall clock durations of the serial, parallel and parallel with queue are summarized in the table below for a 4-core CPU PC and python 3.14.3t:

#	Serial	Parallel	Queue (batch=10)	Queue (batch=100)
1	54.41	18.99	22.25	17.55
2	56.17	17.16	17.72	16.68
3	53.81	17.00	17.24	15.19
4	54.20	17.59	16.51	15.14
5	52.72	17.50	18.20	14.77
6	55.44	17.92	14.85	15.02
7	49.51	17.65	16.96	14.73
8	55.03	17.37	16.70	14.31
9	51.63	17.51	16.30	15.08
10	52.69	17.58	17.29	14.46

4.4.4 Parameter optimization

> **Note**
>
> If you are reading this section just after the introduction on processor workflows, you are not yet familiar with other important concepts of MAFw like the steering file and the possibility to execute your pipelines without having to code a main function. If you want, you can skip this section for now, but remember to come back afterwards!

I know that you are now looking forward to feel the wind of speed thanks to parallel processing, but at the same time you might be feeling afraid of having to test all possible combination of architectures (thread pool vs consumer/producer) and operational parameters (max_workers, queue_size, queue_batch_size…). This might be time consuming especially if you have to edit the code every time to change various parameters.

Have a look at the following piece of code and you will be surprised by how simply the optimization can be performed.

FindPrimeNumberInRangeTunable

```python
class FindPrimeNumberInRangeTunable(LoopTuningMixin, Processor):
    """
    An example processor demonstrating loop tuning via :class:`~mafw.tools.loop_tuning.LoopTuningMixin`.

    The processor can switch between serial and parallel for-loop workflows by setting the
    ``loop_type`` parameter in the steering file, together with the queue and worker settings.
    """

    start_from = ActiveParameter('start_from', default=50, help_doc='From which number to start the search')
    stop_at = ActiveParameter('stop_at', default=100, help_doc='At which number to stop the search')

    def __init__(self, *args: Any, **kwargs: Any):
        """
        Processor parameters:

        :param start_from: First element of the range under investigation.
        :type start_from: int
        :param stop_at: Last element of the range under investigation.
        :type stop_at: int
        """
        super().__init__(*args, **kwargs)
        self.prime_num_found: list[int] = []
        """The list with the found prime numbers."""
        self._lock = threading.Lock()
        """Lock to protect shared list updates in parallel execution."""

    def format_progress_message(self) -> None:
        self.progress_message = (
            f'Checking integer number: {self.item}, already found {len(self.prime_num_found)} prime numbers'
        )

    def get_items(self) -> Collection[Any]:
        """
        Return a range of odd integer numbers between start_from and stop_at.

        :return: A range of odd integer numbers between start_from and stop_at.
        :rtype: range[int]
        """
        if self.start_from >= self.stop_at:
            log.critical(f'{self.start_from} must be smaller than {self.stop_at}')
            return []

        if self.start_from != 2 and self.start_from % 2 == 0:
            self.start_from += 1

        if self.stop_at != 2 and self.stop_at % 2 == 0:
            self.stop_at -= 1

        return range(self.start_from, self.stop_at, 2)
```

(continues on next page)

(continued from previous page)

```
    def start(self) -> None:
        """Reset the list of found prime numbers."""
        super().start()
        self.prime_num_found = []

    def process(self) -> int | None:
        """
        Process a single item and return the prime number payload if found.
        """
        if is_prime(self.item):
            return self.item
        self.looping_status = LoopingStatus.Skip
        return None

    def accept_item(self, result: LoopResult | None = None) -> None:
        """
        Store the prime number once the item has been accepted.

        :param result: The loop result payload produced during processing.
        :type result: mafw.models.LoopResult, Optional
        """
        payload = result.payload if result is not None else self.item
        if payload is None:
            return
        with self._lock:
            self.prime_num_found.append(payload)

    def finish(self) -> None:
        """Print summary information about the prime numbers found."""
        super().finish()
        log.info(
            'Found %s prime numbers in the range from %s to %s'
            % (len(self.prime_num_found), self.start_from, self.stop_at)
        )
        if len(self.prime_num_found):
            log.info('The smallest is %s', min(self.prime_num_found))
            log.info('The largest is %s', max(self.prime_num_found))
```

Steering file

```
# prime_benchmark.toml

processors_to_run = [
    "FindPrimeNumberInRangeTunable#warmup",
    "FindPrimeNumberInRangeTunable#parallel",
    "FindPrimeNumberInRangeTunable#queue_1",
    "FindPrimeNumberInRangeTunable#queue_10",
    "FindPrimeNumberInRangeTunable#queue_100",
    "FindPrimeNumberInRangeTunable#serial",
]

analysis_name = "Parameter tuning"
analysis_description = """PrimeNumber tuning"""

[FindPrimeNumberInRangeTunable] # An example processor demonstrating loop tuning via␣
```

(continues on next page)

(continued from previous page)

```
↪:class:`~mafw.tools.loop_tuning.LoopTuningMixin`.
loop_type = "for_loop" # Loop type to use: for_loop, parallel_for_loop, parallel_for_
↪loop_with_queue.
max_workers = 8 # Number of worker threads for parallel loops.
queue_size = 16 # Queue size for the queue-based parallel loop.
queue_batch_size = 1 # Batch size for queue processing in the queue-based parallel␣
↪loop.
start_from = 9999956789 # From which number to start the search
stop_at = 9999997789 # At which number to stop the search

["FindPrimeNumberInRangeTunable#warmup"] # An example processor demonstrating loop␣
↪tuning via :class:`~mafw.tools.loop_tuning.LoopTuningMixin`.
loop_type = "parallel_for_loop" # Loop type to use: for_loop, parallel_for_loop,␣
↪parallel_for_loop_with_queue.
start_from = 19999956789 # From which number to start the search
stop_at = 19999997789 # At which number to stop the search

["FindPrimeNumberInRangeTunable#parallel"] # An example processor demonstrating loop␣
↪tuning via :class:`~mafw.tools.loop_tuning.LoopTuningMixin`.
loop_type = "parallel_for_loop" # Loop type to use: for_loop, parallel_for_loop,␣
↪parallel_for_loop_with_queue.
max_workers = 8 # Number of worker threads for parallel loops.

["FindPrimeNumberInRangeTunable#queue_1"] # An example processor demonstrating loop␣
↪tuning via :class:`~mafw.tools.loop_tuning.LoopTuningMixin`.
loop_type = "parallel_for_loop_with_queue" # Loop type to use: for_loop, parallel_for_
↪loop, parallel_for_loop_with_queue.
max_workers = 8 # Number of worker threads for parallel loops.
queue_size = 16 # Queue size for the queue-based parallel loop.
queue_batch_size = 1 # Batch size for queue processing in the queue-based parallel␣
↪loop.

["FindPrimeNumberInRangeTunable#queue_10"] # An example processor demonstrating loop␣
↪tuning via :class:`~mafw.tools.loop_tuning.LoopTuningMixin`.
loop_type = "parallel_for_loop_with_queue" # Loop type to use: for_loop, parallel_for_
↪loop, parallel_for_loop_with_queue.
max_workers = 8 # Number of worker threads for parallel loops.
queue_size = 16 # Queue size for the queue-based parallel loop.
queue_batch_size = 10 # Batch size for queue processing in the queue-based parallel␣
↪loop.

["FindPrimeNumberInRangeTunable#queue_100"] # An example processor demonstrating loop␣
↪tuning via :class:`~mafw.tools.loop_tuning.LoopTuningMixin`.
loop_type = "parallel_for_loop_with_queue" # Loop type to use: for_loop, parallel_for_
↪loop, parallel_for_loop_with_queue.
max_workers = 8 # Number of worker threads for parallel loops.
queue_size = 16 # Queue size for the queue-based parallel loop.
queue_batch_size = 100 # Batch size for queue processing in the queue-based parallel␣
↪loop.
```

Let's focus first on the processor implementation. By now you are for sure an expert in all possible ways to search for prime numbers in a range. This implementation is a kind of *Frankenstein-style* mixture between the others we have seen so far. But let's take it one step at the time.

First of all, we do not have any loop type decorator, because we want to be able to change the loop type at runtime. If you do not provide any loop type definition, the processor will follow a standard (serial) for loop workflow. Nevertheless line 1 is still very important because in the inheritance tree, before the `base.Processor` class, our

prime number finder is inheriting from *LoopTuningMixin*, that is what will make the magic.

Line 24 is where we define a thread safe lock to protect the insertion in the shared result list. This is completely harmful in the case of serial and queue-based for loop, because being the list accessed by a single thread, the lock will also be open.

The implementation of the process method is identical as in the case of *FindPrimeNumberInRangeParallelWithQueue* with a well defined return statement (line 61) where we channel the process result to the accept/skip item part. In the case of serial and thread-pool-based for loop, this is not strictly required because the two blocks are living in the same thread, but it is important for the queue based approach.

The accept_item is a mixture of the *FindPrimeNumberInRangeParallelWithQueue* and *FindPrimeNumberInRangeParallel* implementations, where we receive a payload (for queue based) or we use the item attribute (for serial and parallel) and we append it to the shared list of results protected by the lock.

Ok, now that the code is clear, let's have a look at the steering file in the second tab. There are 6 processors in the pipeline. They are all of FindPrimeNumberInRangeTunable type with different replica names. The first one (*warm-up*) is using the parallel_for_loop to cover the standard 20500 integer numbers but starting from a much higher initial value compared to the previous example. We run this first to be sure that all CPU cores are throttling at the maximum speed and we will discard its output. Then we start with the real experiment on the range of interest. The first is parallel mode, then queue based with different batch sizes (1, 10 and 100) and finally the serial loop for reference (it will use the base processor parameter definitions).

And now, you open your console and run the pipeline *mafw run prime_benchmark.toml*. At the end of each processor, have a look at the timing statistics. In the table below, the output of a 4-core PC.

Step	Duration
Warmup	31.3 s
Parallel	19.8 s
Queue 1	19.4 s
Queue 10	16.5 s
Queue 100	15.1 s
Serial	56.6 s

This very fast *hyper-parameter tuning* (allow me to use this expression borrowed from machine learning) demonstrate the way to go: for this interval the queue based approach is better than the parallel and we can observe a clear improvement increasing the batch size.

Now that you know what is the right looping strategy and the most performing parameters, you can (**if you wish - you do not have to!**) go back to your code and fix the value in the processor constructor like this:

```
@parallel_for_loop_with_queue
class FindPrimeNumberInRangeTunable(Processor):
    # keep the rest of the code as it was, just change the highlighted lines

    def __init__(self, *args: Any, **kwargs: Any):
        """
        Processor parameters:

        :param start_from: First element of the range under investigation.
        :type start_from: int
        :param stop_at: Last element of the range under investigation.
        :type stop_at: int
        """
        super().__init__(*args,  queue_batch_size=100, **kwargs)
        self.prime_num_found: list[int] = []
        """The list with the found prime numbers."""
```

(continues on next page)

(continued from previous page)

```
17         self._lock = threading.Lock()
18         """Lock to protect shared list updates in parallel execution."""
```

4.5 Importing elements to the database

> **Note**
>
> This example is using concepts that have not yet been introduced, in particular the database. So in a first instance, you can simply skip it and come back later.

Importing elements in the database is a very common task, that is required in all analytical projects. To accomplish this task, mafw is providing a dedicated base class (the *Importer*) that heavily relies on the use of the *FilenameParser* to extract parameters from the filenames.

The *ImporterExample* is a concrete implementation of the base Importer that can be used by the user to get inspiration in the development of their importer subclass.

Before diving into the *ImporterExample* code analysis, we should understand the role and the functionality of other two helper classes: the *FilenameElement* and the *FilenameParser*.

4.5.1 Retrieving information from filenames

When setting up an experimental plan involving the acquisition of several data files, there are different approaches.

1. The **descriptive** approach, where the filename is used to store information about the measurement itself,
2. the **metadata** approach, where the same information are stored inside the file in a metadata section,
3. or the **logbook** approach, where the filename is just a unique identifier and the measurement information are stored in a logbook (another file, database, piece of paper…) using the same unique identifier.

The **descriptive** approach, despite being sometime a bit messy because it may end up with very long filenames, it is actually very practical. You do not need to be a hacker including the metadata in the file itself and you do not risk to forget to add the parameters to the logbook.

The tricky part is to include those information to the database containing all your experiments, and you do not want to do this by hand to avoid errors.

The best way is to use regular expression that is a subject in which python is performing excellently and MAFw is helping you with two helpers.

The first helper is the *FilenameElement*. This represents one single piece of information that is stored in the filename.

Let us assuming that you have a file named as `sample_12_energy_10_repetition_2.dat`. You can immediately spot that there are three different pieces of information stored in the filename. The sample name, the value of the energy in some unit that you should known, and the value of the repetition. Very likely there is also a repetition_1 file saved on disc.

In order to properly interpret the information stored in the filename, we need to define three *FilenameElement* s, one for each of them!

If you look at the documentation of the *FilenameElement*, you will see that you need four arguments to build it:

- its **name**, this is easy. Take one, and use it to name a named group in the regular expression.
- its **regular expression**, this is tricky. This is the pattern that python is using to read and parse the actual element.
- its **type**, this is the expected type for the element. It can be a string, an integer or a floating point number.

- its **default** value, this is used to make the element optional. It means that if the element is not found, then the default value is returned. If no default value is provided and the element is not found then an error is raised.

Let us see how you could use *FilenameElement* class to parse the example filename.

```
filename = 'sample_12_energy_10_repetition_2.dat'

sample = FilenameElement('sample', r'[_]*(?P<sample>sample_\d+)[_]*', value_type=str)
energy = FilenameElement('energy', r'[_]*energy_(?P<energy>\d+\.*\d*)[_]*', value_
↪type=float)
repetition = FilenameElement(
    'repetition', r'[_]*repetition_(?P<repetition>\d+)[_]*', value_type=int, default_
↪value=1
)

sample.search(filename)
assert sample.value == 'sample_12'

energy.search(filename)
assert energy.value == 10

repetition.search(filename)
assert repetition.value == 2
```

The interesting thing is that you can swap the position of the elements in the filename, for example starting with the energy, and it will still be working absolutely fine.

Just open a python interpreter, import the FilenameElement class and give it a try yourself to familiarize with the regular expression. Be careful, when you write the regular expression pattern, since it usually contains a lot of '\', it may be useful to prefix the string with a r, in order to inform python that what is coming must be interpreted as a raw string.

If you want to gain confidence with regular expressions, make some tests and understand their power, we recommend to play around with one of the many online tools available on the web, like pythex.

The *FilenameElement* is already very helpful, but if you have several elements in the filename, the readability of your code will quickly degrade. To help you further, you can enjoy the *FilenameParser*.

This is actually a combination of filename elements and when you will try to interpret the filename by invoking *interpret()* all of the filename elements will be parsed and thus you can retrieve all parameters in a much easier way.

If you look at the *FilenameParser* documentation, you will see that you need a configuration file to build an instance of it. This configuration file is actually containing the information to build all the filename element.

In the two tabs here below you can see the configuration file and the python code.

Parser configuration

```
# FilenameParser configuration file
#
# General idea:
#
# The file contains the information required to build all the FilenameElement␣
↪requested by the importer.
#
# Prepare a table for each element and in each table add the regexp, the type and␣
↪optionally the default.
# Adding the default field, will make the element optional.
#
# Add the table name in the elements array. The order is irrelevant. The division in␣
```

(continues on next page)

(continued from previous page)

```
↪compulsory and optional elements
# is also irrelevant. It is provided here just for the sake of clarity.
#
# You can have as many element tables as you like, but only the one listed in the␣
↪elements array will be used to
# configure the Importer.
#
elements = [
    # compulsory elements:
    'sample', 'energy',
    # optional elements:
    'repetition'
]


[sample]
regexp = '[_]*(?P<sample>sample_\d+)[_]*'
type='str'

[energy]
regexp = '[_]*energy_(?P<energy>\d+\.*\d*)[_]*'
type='float'

[repetition]
regexp = '[_]*repetition_(?P<repetition>\d+)[_]*'
type='int'
default = 1
```

Python test code

```
filename = 'energy_10.3_sample_12.dat'

parser = FilenameParser('example_conf.toml')
parser.interpret(filename)

assert parser.get_element_value('sample') == 'sample_12'
assert parser.get_element_value('energy') == 10.3
assert parser.get_element_value('repetition') == 1
```

The configuration file must contain a top level `elements` array with the name of all the filename elements that are included into the filename. For each value in `elements`, there must be a dedicated table with the same name containing the definition of the regular expression, the type and optionally the default value.

Important

In TOML configuration files, the use of single quotation marks allows to treat a string as a raw string, that is very important when passing expression containing backslashes. If you prefer to use double quotation marks, then you have to escape all backslashes.

The order of the elements in the `elements` array is irrelevant and also the fact we have divided them in compulsory and optional is just for the sake of clarity.

In the python tab, you can see how the use of *FilenameParser* makes your code looking much tidier and easier to read. In this second example, we have removed the optional specification of the `repetition` element and you can see that the parser is returning the default value of 1 for such element and we have swapped the energy field with the sample name. Moreover, now the energy field is actually a floating number with a decimal figure.

4.5.2 The basic importer

With the power of these two helper classes, building a processor for parsing all our measurement filenames is a piece of a cake. In the *processor_library* package, you can find a basic implementation of a generic *Importer* processor, that you can use as a base class for your specific importer.

The idea behind this importer is that you are interested in files inside an `input_folder` and possibly all its subfolders. You can force the processor to look recursively in all subfolder by turning the processor parameter `recursive` to True. The last parameter of this processor is the `parser_configuration` that is the path to the *FilenameParser* configuration file.

This configuration file is used during the *start()* method of *Importer* (or any of its subclasses) to configure its *FilenameParser*, so that you do not have to worry of this step. In your subclass process method, the filename parser will be straight away ready to use.

Let us have a loop and the *ImporterExample* processor (available in the *examples* package) for a concrete implementation of an importer processor.

4.5.3 The ImportExample processor

We will build a subclass of the *Importer* processor following the *for_loop* execution workflow.

In the *start()* method, we will assure that the target table in the database is existing. The definition of the target database Model (*InputElement* in this example) should be done in a separate database model module to facilitate import statements from other modules as well.

```
class InputElement(MAFwBaseModel):
    """A model to store the input elements"""

    element_id = AutoField(primary_key=True, help_text='Primary key for the input␣
↪element table')
    filename = FileNameField(unique=True, checksum_field='checksum', help_text='The␣
↪filename of the element')
    checksum = FileChecksumField(help_text='The checksum of the element file')
    sample = TextField(help_text='The sample name')
    exposure = FloatField(help_text='The exposure time in hours')
    resolution = IntegerField(default=25, help_text='The readout resolution in µm')
```

```
def start(self) -> None:
    """
    The start method.

    The filename parser is ready to use because it has been already configured in the␣
↪super method.
    We need to be sure that the input table exists, otherwise we create it from␣
↪scratch.
    """
    super().start()
    self.database.create_tables([InputElement])
```

In the *get_items()*, we create a list of all files, in this case matching the fact that the extension is *.tif*, included in the `input_folder`. We use the `recursive` flag to decide if we want to include also all subfolders.

The *steering* file may contain a GlobalFilter section (see *the Filter section*) and we use the new_only flag of the filter_register, to further filter the input list from all files that have been already included in the database. It is also important to check that the table is update because you may have an entry pointing to the same filename that in the mean time has been modified. For this purpose the *verify_checksum()* can be very useful. A more detailed explanation of this function will be presented in a *subsequent section*.

```
def get_items(self) -> Collection[Any]:
    r"""Retrieves the list of element to be imported.
```

(continues on next page)

(continued from previous page)

```
    The base folder is provided in the configuration file, along with the recursive␣
↪flags and all the filter options.

    :return: The list of items full file names to be processed.
    :rtype: list[Path]
    """
    pattern = '**/*tif' if self.recursive else '*tif'
    input_folder_path = Path(self.input_folder)

    file_list = [file for file in input_folder_path.glob(pattern) if file.is_file()]

    # verify the checksum of the elements in the input table. if they are not up to␣
↪date, then remove the row.
    verify_checksum(InputElement)

    if self.filter_register.new_only:
        # get the filenames that are already present in the input table
        existing_rows = InputElement.select(InputElement.filename).namedtuples()
        # create a set with the filenames
        existing_files = {row.filename for row in existing_rows}
        # filter out the file list from filenames that are already in the database.
        file_list = [file for file in file_list if file not in existing_files]

    return file_list
```

The *ImporterExample* follows an implementation approach that tries to maximise the efficiency of the database transaction. It means that instead of making one transaction for each element to be added to the database, all elements are collected inside a list and then transferred to the database with a cumulative transaction at the end of the process itself. This approach, as said, is very efficient from the database point of view, but it can be a bit more demanding from the memory point of view. The best approach depends on the typical number of items to be added for each run and the size of each element.

The implementation of the *process()* is rather simple and as you can see from the source code it is retrieving the parameter values encoded in the filename via the *FilenameParser*. If you are wondering why we have assigned the filename to the filename and to the checksum field, have a look at the section about *custom fields*.

```
def process(self):
    """
    The process method overload.

    This is where the whole list of files is scanned.

    The current item is a filename, so we can feed it directly to the FilenameParser␣
↪interpret command, to have it
    parsed. To maximise the efficiency of the database transaction, instead of␣
↪inserting each file
    singularly, we are collecting them all in a list and then insert all of them in␣
↪the :meth:`~.finish` method.

    In case the parsing is failing, then the element is skipped and an error message␣
↪is printed.
    """
    try:
        new_element = {}
        self._filename_parser.interpret(self.item.name)
        new_element['sample'] = self._filename_parser.get_element_value('sample_name')
```

(continues on next page)

(continued from previous page)

```
        new_element['exposure'] = self._filename_parser.get_element_value('exposure')
        new_element['resolution'] = self._filename_parser.get_element_value(
↪'resolution')
        new_element['filename'] = self.item
        new_element['checksum'] = self.item
        self._data_list.append(new_element)
    except ParsingError:
        log.critical('Problem parsing %s' % self.item.name)
        self.looping_status = LoopingStatus.Skip
```

The *finish()* is where the real database transaction is occurring. All the elements have been collected into a list, so we can use an insert_many statement to transfer them all to the corresponding model in the database. Since we have declared the filename field as unique (this was our implementation decision, but the user is free to relax this requirement), we have added a `on_conflict` clause to deal with the case the user is updating an entry with the same filename.

Since the *super method* is printing the execution statistics, we are leaving its call at the end of the implementation.

```
def finish(self) -> None:
    """
    The finish method overload.

    Here is where we do the database insert with a on_conflict_replace to cope with␣
↪the unique constraint.
    """
    # we are ready to insert the lines in the database
    InputElement.insert_many(self._data_list).on_conflict_replace(replace=True).
↪execute()

    # the super is printing the statistics, so we call it after the implementation
    super().finish()
```

CHAPTER 5

PROCESSORLIST: COMBINE YOUR PROCESSORS IN ONE GO

So far we have seen how a *Processor* can be coded to perform a simple task with a certain degree of generality offered by the configurable parameters. But analytical tasks are normally rather complex and coding the whole task in a single processor will actually go against the mantra of simplicity and code reusability of MAFw.

To tackle your complex analytical task, MAFw proposes a solution that involves chaining multiple processors together. The following processors can start where the previous one stopped so that like in a building game, scientists can put together their analytical solution with simple blocks.

From a practical point of view, this is achieved via the *ProcessorList* that is an evolution of the basic python list, which can contain only instances of processor subclasses or other ProcessorLists.

Once you have appended the processors in the order you want them to be executed, just call the *execute()* method of the list and it will take care of running all the processors.

As simple as that:

```
def run_simple_processor_list():
    """Simplest way to run several processors in a go."""
    from mafw.examples.sum_processor import AccumulatorProcessor, GaussAdder
    from mafw.processor import ProcessorList

    # create the list. name and description are optional
    new_list = ProcessorList(name='AddingProcessor', description='Summing up numbers')

    # append the processors. you can pass parameters to the processors in the␣
↪standard way
    max_value = 120
    new_list.append(AccumulatorProcessor(last_value=max_value))
    new_list.append(GaussAdder(last_value=max_value))

    # execute the list. This will execute all the processors in the list
    new_list.execute()

    # you can access single processors in the list, in the standard way.
    # remember that the ProcessorList is actually a list!
    assert new_list[0].accumulated_value == new_list[1].sum_value
```

5.1 The ProcessorExitStatus

We have seen in a *previous section* that the user can modify the looping behavior of a processor by using the *LoopingStatus* enumerator. In a similar manner, the execution loop of a processor list can be modified looking at the *ProcessorExitStatus* of each processors.

When one processor in the list is finishing its task, the *ProcessorList* is checking for its exit status before moving to the next item. If a processor is finishing with an Abort status, then the processor list will raise a *AbortProcessorException* that will cause the loop to be interrupted.

Let us have a look at the snippet here below:

```
def run_processor_list_with_loop_modifier():
    """Example on deal with processors inside a processor list changing the loop␣
↪structure.

    In this example there are two processors, one that will run until the end and the␣
↪other that will set the looping
    status to abort half way. The user can see what happens when the :class:`~mafw.
↪processor.ProcessorList` is executed.
    """
    import time

    from mafw.enumerators import LoopingStatus, ProcessorExitStatus, ProcessorStatus
    from mafw.mafw_errors import AbortProcessorException
    from mafw.processor import ActiveParameter, Processor, ProcessorList

    class GoodProcessor(Processor):
        n_loop = ActiveParameter('n_loop', default=100, help_doc='The n of the loop')
        sleep_time = ActiveParameter('sleep_time', default=0.01, help_doc='So much␣
↪work')

        def get_items(self) -> list[int]:
            return list(range(self.n_loop))

        def process(self):
            # pretend to do something, but actually sleep
            time.sleep(self.sleep_time)

        def finish(self):
            super().finish()
            print(f'{self.name} just finished with status: {self.processor_exit_
↪status.name}')

    class BadProcessor(Processor):
        n_loop = ActiveParameter('n_loop', default=100, help_doc='The n of the loop')
        sleep_time = ActiveParameter('sleep_time', default=0.01, help_doc='So much␣
↪work')
        im_bad = ActiveParameter('im_bad', default=50, help_doc='I will crash it!')

        def get_items(self) -> list[int]:
            return list(range(self.n_loop))

        def process(self):
            if self.item == self.im_bad:
                self.looping_status = LoopingStatus.Abort
                return
            # let me do my job
```

(continues on next page)

(continued from previous page)

```
41              time.sleep(self.sleep_time)
42
43          def finish(self):
44              super().finish()
45              print(f'{self.name} just finished with status: {self.processor_exit_
↪status.name}')
46
47      proc_list = ProcessorList(name='with exception')
48      proc_list.extend([GoodProcessor(), BadProcessor(), GoodProcessor()])
49      try:
50          proc_list.execute()
51      except AbortProcessorException:
52          print('I know you were a bad guy')
53      assert proc_list.processor_exit_status == ProcessorExitStatus.Aborted
54      assert proc_list[0].processor_exit_status == ProcessorExitStatus.Successful
55      assert proc_list[1].processor_exit_status == ProcessorExitStatus.Aborted
56      assert proc_list[2].processor_status == ProcessorStatus.Init
```

We created two processors, a good and a bad one. The good one is doing nothing, but getting till the end of its job. The bad one is also doing nothing but giving up before the end of the item list. In the process method, the bad processor is setting the looping status to abort, causing the for loop to break immediately and to call finish right away. In the processor finish method, we check if the status was aborted and in such a case we set the exit status of the processor to Aborted.

At line 47, we create a list and we populate it with three elements, a good, a bad and another good processor and we execute it inside a try/except block. The execution of the first good processor finished properly as you can see from the print out and also from the fact that its status (line 54) is Successful. The second processor did not behave, the exception was caught by the except clause and this is confirmed at line 55 by its exit status. The third processor was not even started because the whole processor list got stopped in the middle of processor 2.

5.2 Resources acquisition and distribution

While it may seems somewhat technical for this for this tutorial, it is worth highlighting an intriguing implementation detail. If you look at the constructor of the *Processor* class, you will notice that you can provide some resources, like the Timer and UserInterface even though we have never done this so far. The idea is that when you execute a single processor, it is fully responsible of creating the required resources by itself, using them during the execution and then closing them when finishing.

Just as an example, consider the case of the use of a database interface. The processor is opening the connection, doing all the needed transactions and finally closing the connection. This approach is also very practical because it is much easier to keep expectations under control.

If you run a *ProcessorList*, you may want to move the responsibility of handling the resources from the single Processor to the output ProcessorList. This approach allows all processors to share the same resources efficiently, eliminating the need to repeatedly open and close the database connection each time the ProcessorList advances to the next item.

You do not have to care about this shift in responsibility, it is automatically done behind the scene when you add a processor to the processor lists.

5.3 What's next

In this part we have seen how we can chain the execution of any number of processors all sharing the same resources. Moreover, we have seen how we can change the looping among different processors using the exit status.

Now it is time to move forward and see how you can add your own processor library!

CHAPTER 6

PLUGINS: IMPORT YOUR PROCESSORS

We are almost half-way through our tutorial on MAFw. We have learned what a *Processor* is and how we can create our own processors just by subclassing the base class. With the *ProcessorList* we have seen an easy way to chain the execution of many processors. So by now, you might be tempted to open your IDE and start coding stuff… But please hold on your horses for another minute, MAFw has much more to offer.

In the previous pages of this tutorial, we always coded our *processors* and then executed them manually, either from the python console or via an ad-hoc script. This is already helpful, but not really practical, because everytime you want to change which *processors* are executed or one of their parameters, we need to change the script code. This seems very likely the job for a configuration file, doesn't it?

MAFw is providing you with exactly this capability, a generic script that will read your configuration file (**steering file** in the MAFw language) and execute it. Here comes the problem, *how can the execution framework load your processors since they are not living in the same library*?

To this problem there are two solutions: a quick and dirty and a clean and effective! Of course you will have the tendency to prefer the first one, but really consider the benefits of the second one before giving up reading!

6.1 The quick and dirty: code inside MAFw

When you install MAFw on your system, you can always do it in development mode (for `pip` this corresponds to the *-e* option). In this way, you will be allowed to code your *processors* library directly inside your locally installed MAFw.

It is very quick, because you can start coding right away, but it is also rather dirty, because it will be much harder to install MAFw updates and after a while the project will become too big and messy.

For these and many other reasons, we are convinced that this is not the right way to go.

6.2 The clean and effective: turn your library into a plugin

It may seem more challenging than it actually is, but in reality, it is quite simple. MAFw uses the plugin system developed by pluggy, that is very powerful and at the same time relatively easy to deploy.

After you have installed MAFw in your system (you can do it in development mode if you want to contribute!), create a new project to store your processor. How to create a project is well described in details here with a step by step guide. You don't need to upload your package tp PyPI, you can stop as soon as you are able to build a wheel of your package and install it in a virtual environment.

To achieve this, you would need to have a *pyproject.toml* file, with the project metadata, the list of dependencies and other things. Here below is an example of what you should have:

Listing 6.1: An example of pyproject.toml for a plugin

```
[build-system]
requires = ["hatchling"]
build-backend = "hatchling.build"

[project]
name = "fantastic_analysis"
dynamic = ["version"]
description = 'My processor library'
readme = "README.md"
requires-python = ">=3.8"
license = "MIT"
keywords = []
authors = [
  { name = "Surname Name", email = "this.is.me@my.domain.com" },
]
classifiers = [
  "Development Status :: 4 - Beta",
  "Programming Language :: Python",
  "Programming Language :: Python :: 3.8",
  "Programming Language :: Python :: 3.9",
  "Programming Language :: Python :: 3.10",
  "Programming Language :: Python :: 3.11",
  "Programming Language :: Python :: 3.12",
  "Programming Language :: Python :: Implementation :: CPython",
  "Programming Language :: Python :: Implementation :: PyPy",
]

dependencies = ['mafw'] # plus all your other dependencies

[project.urls]
Documentation = "https://github.com/..."
Issues = "https://github.com/..."
Source = "https://github.com/."

## THIS IS THE KEY PART
## --------------------
# you can add as many lines you want to the table.
# always use unique names for the entry.

[project.entry-points.'mafw']
fantastic_analysis_plugin = 'fantastic_analysis.plugins'

[tool.hatch.version]
path = "src/fantastic_analysis/__about__.py"
```

Particularly important are lines 40 and 41. There you declare that when installed in an environment, your package is providing a plugin for MAFw, in particular this plugin that you named *fantastic_analysis_plugin* is located inside your package in a file named plugins.py.

So now, let us have a look at what do you have to have inside this file:

Listing 6.2: Example of plugins.py

```
"""
This is the module that will be exposed via the entry point declaration.

```

(continues on next page)

(continued from previous page)

```
Make sure to have all processors that you need to export in the list.
"""

import mafw
from mafw.processor import Processor
from fantastic_analysis import my_processor_lib

@mafw.mafw_hookimpl
def register_processors() -> list[mafw.processor.Processor]:
    return [my_processor_lib.Processor1, my_processor_lib.Processor2]
```

In this python file you need to import mafw in order to have access to the *mafw_hookimpl* decorator. This is a marker for pluggy so that it knows that this function should return something for the host application. The second import statement is needed only for the typing of the return value. The third one is to import your library with the processors. It can be one or as many as you have.

Inside the returned list, just put all the processors you want to export and the trick is done!

Now install in development mode your package and MAFw will be able to access all your processor library, without you having to do anything else!

6.2.1 Using a proxy instead of a real processor

Exporting all your processor is as easy as listing them in the *register_processors* hook. There is a limitation with this approach and it becomes clear as soon as you have many processors. Imagine to have a dozen of processors, each of them importing large modules (pandas, matplotlib, TensorFlow…). When MAFw will retrieve the processor list from your plugin, it will have to import your modules triggering the import of all other dependencies. In other words, your calculation have not started yet and all the modules are imported, including the ones that are needed by processors that will not be executed!

The solution is to use a so-called *lazy proxy*, it is to say an object that will behave like the real processor class during the plugin discovery, but it will reify in the real processor when it is needed! If you are interested in the implementation, have a look at the proxy *API*.

The performance gain following this approach is really impressive and it cost you nothing. The code below is totally equivalent to the *previous one*, but faster!

Listing 6.3: Example of plugins.py with proxy

```
"""
This is the module that will be exposed via the entry point declaration.

Make sure to have all processors that you need to export in the list.
"""
import mafw
from mafw.lazy_import import LazyImportProcessor, ProcessorClassProtocol

@mafw.mafw_hookimpl
def register_processors() -> list[ProcessorClassProtocol]:
    return [
        LazyImportProcessor('fantastic_analysis.my_processor_lib', 'Processor1'),
        LazyImportProcessor('fantastic_analysis.my_processor_lib', 'Processor2')
    ]
```

Migration from MAFw v1.4

If you are migrating from v1.4 to v2, then you may have noticed that the plugin hook to export the standard tables has disappeared.

If fact, the standard table plugin became useless with the introduction of the `ModelRegister`, where all database Models are stored automatically. The same applies to models inheriting to `StandardTable`, so you can easily retrieve them via the `model register`.

Refer to these two other paragraphs: *Automatic model registration* and *Standard tables*.

6.3 What's next

If you are still reading this tutorial, it means that you are aware of the benefits that MAFw can bring to your daily analysis task. And the best part is, there is even more to discover! On the following section, you will discover how to efficiently manage your complex analytical process by simply listing all the sequential steps in a steering file.

CHAPTER 7

RUN YOUR PROCESSORS FROM THE COMMAND LINE

Now that you have a wonderful library of *processors* that are exported as plugins to MAFw, you can benefit from an additional *bonus*, the possibility to execute them without having to code any other line!

When you have installed *MAFw* in your environment, an executable file mafw has been put in your path, so that if you open a console and you activate the python environment, you can run it.

Just try something like this!

```
(env) C:\>mafw -v
mafw, version 0.0.5
```

If you get this message (the version number may vary), you are good to go. Otherwise, check that everything was installed correctly and try again.

This executable is rather powerful and can launch several commands (see the full documentation *here*), but for the time we will focus on three.

7.1 List the processors

Before doing anything else, it is important to check that your processor library has been properly loaded in MAFw. To check that, run the following command from the console.

```
(env) C:\>mafw list

                    Available processors (External Plugins = True)
                ┌──────────────────────┬──────────────────┬──────────────────────┐
                │ Processor Name       │ Package Name     │ Module               │
                ├──────────────────────┼──────────────────┼──────────────────────┤
                │ AccumulatorProcessor │ mafw             │ examples.sum_processor │
                │ GaussAdder           │ mafw             │ examples.sum_processor │
                │ ModifyLoopProcessor  │ mafw             │ examples.loop_modifier │
                │ PrimeFinder          │ mafw_plugin_test │ prime_finder           │
                └──────────────────────┴──────────────────┴──────────────────────┘
                    Total processors = 4, internal = 3, external = 1
```

The output should be a table similar to the one above, containing a list of known processors in the first column, the package where they come from in the second and the module in the third one.

If you have exported your processors correctly the total external processors should be different from zero.

7.2 Prepare a generic steering file

The next step is to prepare a steering file. We do not want you to open your text editor and start preparing the TOML file from scratch. MAFw can prepare a generic template for you that you have to adapt to your needs before running.

```
(env) C:\>mafw steering steering-file.toml
A generic steering file has been saved in steering-file.toml.
Open it in your favourite text editor, change the processors_to_run list and save it.

To execute it launch: mafw run steering-file.toml.
```

This command is generating a generic steering file with a list of all processors available (internal to MAFw and imported from your library) with all their configurable parameters. You can pass the option *–show* to display the generated file on the console, or even more practical the *–open-editor* to open the steering file in your default editor so that you can immediately customize it.

Here below is an example of what you will get. The content may vary, but the ideas are the same.

Listing 7.1: An example of steering file.

```
1  # MAFw steering file generated on 2024-11-21 16:54:22.455297
2
3  # uncomment the line below and insert the processors you want to run from the␣
   ↪available processor list
4  # processors_to_run = []
5
6  # customise the name of the analysis
7  analysis_name = "mafw analysis"
8  analysis_description = "Summing up numbers"
9  available_processors = ["AccumulatorProcessor", "GaussAdder", "ModifyLoopProcessor"]
10
11 [AccumulatorProcessor] # A processor to calculate the sum of the first n values via a␣
   ↪looping approach.
12 last_value = 100 # Last value of the series
13
14 [GaussAdder] # A processor to calculate the sum of the first n values via the so␣
   ↪called *Gauss formula*.
15 last_value = 100 # Last value of the series.
16
17 [ModifyLoopProcessor] # Example processor demonstrating how it is possible to change␣
   ↪the looping structure.
18 item_to_abort = 65 # Item to abort
19 items_to_skip = [12, 16, 25] # List of items to be skipped.
20 total_item = 100 # Total item in the loop.
21
22 [UserInterface] # Specify UI options
23 interface = "rich" # Default "rich", backup "console"
```

The TOML file contains some tables and arrays that should be rather easy to understand. There is an important array at line 4 that you need to uncomment and to fill in with the *processors* that you want to run. For all known processors, there is table with all the configurable parameters, you can change them to your wishes. You do not need to remove the configuration information of the processors that you do not intend to use. They will be simply ignored.

If you have commented your *Processor* classes properly, the short description will appear next to the table.

You can customize the analysis name and short description to make the execution of the steering file a bit nicer. In the UserInterface section, you can change the way the processor will interact with you. If you are running the steering file on a headless HPC, then you do not need to have any fancy output. If you like to see progress bars and

spinners moving, then select rich as your interface.

7.2.1 Environment variable expansion

Standard TOML parsers do not interpolate variables. MAFw extends the steering file parsing step with *environment variable expansion*, so any string value can reference runtime variables.

The expansion follows standard bash-style grammar:

```
output_path = "${DATA_DIR}/results"
cache_dir = "${CACHE_DIR:-/tmp/mafw-cache}"
db_url = "${DB_URL:?Define DB_URL before running}"
```

This makes steering files portable across machines or batch systems, because the same file can adapt to different file layouts or execution environments without manual edits.

You do not like the idea of opening a text editor and manually change the values of the processor parameters? No worries, we have you covered and developed for you a very *user friendly GUI tool* to create valid steering files without having to get crazy with TOML syntax. But before jumping to the GUI user guide, finish this session!

7.3 Run the steering file

Here we are, finally! Now we can run our analysis and enjoy the first results.

From the console type the following command:

```
(env) C:\>mafw run steering-file.toml
```

and enjoy the thrill of seeing your data being processed. From now on, you have a new buddy helping you with the analysis of your data, giving you more time to generate nice and interesting plots because MAFw will do all the boring stuff for you!

7.4 Debugging your processors

When you execute your processors from the command line using *mafw* and something does not work as expected, you may want to debug your processors to fix the problem. Normally you would turn back to your IDE and click on the debug run button to follow your code line by line and see where it fails.

The problem here is that the module where you have coded your processor is not *running*, it just contains the definition of the processor! You would need to create an instance of your processor and to execute it in a standalone way, but then you would have to set the parameters manually. In other words, it is going to be slow and problematic and MAFw wants to alleviate your problems, not to create new ones!

The best way is to run the mafw executable in debug mode from your IDE inserting a breakpoint in your processor code.

In PyCharm, you would create a new python configuration launching a module instead of a script. See the screenshot below.

The three most important points have been highlighted. First of all, you need to select module and not script from the drop down menu indicated by the green arrow. In the text field next to it (yellow in the screenshot), write *mafw.scripts.mafw_exe*. MAFw is installed in the environment where your plugins are living, so PyCharm will complete the module name while writing. In the text field below (orange in the screenshot) write the command line options as you were doing from the console. Optionally, if you want to enjoy the colorful output of the `RichInterface`, click on the *Modify options* (top blue arrow) and add *Emulate terminal in output console*. The corresponding label will appear in the bottom part as indicated by the blue arrow. The disadvantage of this decision is that the console will become readonly, so you want be able to enter python statement while the code execution is stopped.

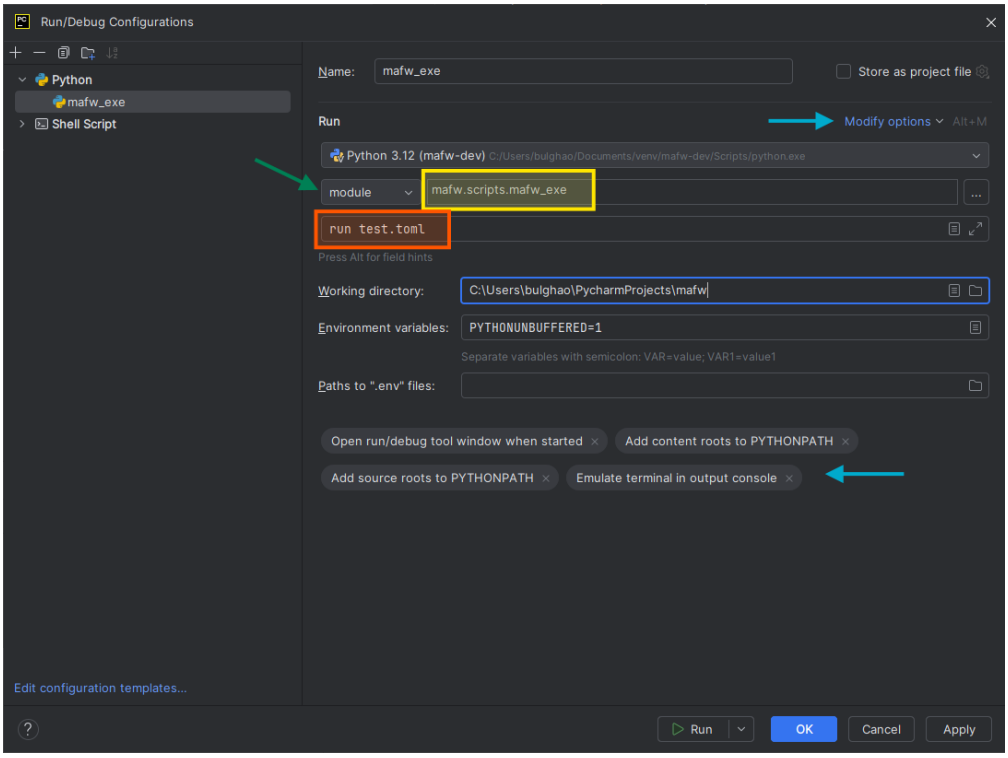


Fig. 7.1: The PyCharm configuration for running mafw_exe from your plugin module. It can be particularly helpful to execute your processor in debug mode.

7.5 Running more instances of the same processor

Can I run the same processor twice? Of course, you can. MAFw is here to facilitate your life, not to make it even more difficult. In the steering file, you set the list of processor to be executed in the array *processors_to_run*. The execution workflow will follow exactly the order you enter the processor in the list. If you want to run the same processor twice, or multiple times, just put its name in the list in the position you want to run it.

If you want your multiple instances to use different processor parameter values, then have a look at the steering file below:

```
processors_to_run = ['MyProcessor', 'MyProcessor#1', 'MyProcessor#2']

[MyProcessor]
param1 = 'base_value'
param2 = 10

["MyProcessor#1"] # note the quotation around the name
param1 = 'value_1'
```

The processors_to_run array contains three elements, the first is our processor, while the other two are still our processors but we have added a # followed by a number (actually it could have been any other string, not necessarily a number). In MAFw terminology, the string after the # is called the *replica id* and it is used to identify uniquely a processor inside the steering file.

When *mafw* is executing the steering file, it will parse the elements in the processor_to_run array and instantiate each processors using the base name (the part on the left of the # sign). During the parameter configuration, each processor will look for a section with either its base name (MyProcessor in this case) or its *so-called replica aware name* (MyProcessor#1 and MyProcessor#2 in this example). By default, the parameter configuration follows a kind of *inheritance scheme*. It means that when a processor is listed with its replica_name, take 'MyProcessor#1' for example, the processor configuration will first load the base configuration and then update it with the replica specific one. In the case of MyProcessor#1, this will result in the value of param1 being set to 'value_1',

while param2 will be 10. The last processor, 'MyProcessor#2' does not have a replica specific section, so it will use the parameter settings as defined in the base section.

> **Warning**
>
> TOML allows to have only alphanumeric digit plus dashes and underscores in a bare key. This means that section for replicas aware processors must be quoted otherwise the # will trigger a parsing error.

If you do not want to have this inheritance between the replica and the base configuration, you can switch it off for a single replica setting the keyword *__inheritance__* to false in its table like this:

```
["MyProcessor#2"]
__inheritance__ = false
```

In this case, the processor configuration will only use the parameters defined in the replica specific table. For all missing parameters, the default value defined in the code with of the *processor.ActiveParameter* definition.

7.6 Grouping processors together

Another interesting features of the MAFw steering file is the possibility to join processors in groups, so that they will be added to the main *processor.ProcessorList* as a nested *processor.ProcessorList*. Imagine, for example, that in your pipeline you have several processors (or processor replicas) all dedicated to generate simulated data, and another group focused on the analysis of real data. You can then prepare a steering file like this:

```
processors_to_run = [ 'Simulation', 'RealData']
analysis_name = 'full-pipeline'
analysis_description = """Full pipeline"""

[Simulation]
name = 'simulation-pipeline' # optional
description = 'Simulation pipeline' # optional
processors_to_run = ['SimulProc#123', 'SimulProc1', 'PlotProcessor#Simul'] #␣
↪compulsory

[RealData]
name = 'real-data-pipeline'
description = 'Real data pipeline'
processors_to_run = ['RealDataProc', 'AnotherProcessor', 'PlotProcessor#Real']

# do not forget to add a section for each of the processors
```

This steering file will run the main processor list containing two other processor lists. The advantage is that if you decide to skip the simulation, you just have to remove one single item from the main processors_to_run variable. For each processor group you can optionally add a name and a description, while the processors_to_run list must be present all the time. You can even add groups to groups, the only thing you cannot do is do add a cyclic recursion; in such a case, MAFw will detect it and raise an error.

7.7 The exit code

The MAFw executable is always releasing an exit code when its execution is terminated. This exit code follows the standard convention of being 0 if the process was successful. Any other number is a symptom that something went wrong during the execution.

You will get an exit code different from 0 whenever, for example, you have not provided all the required parameters or they are invalid. You can also get an exit code different from 0 if your processor failed its execution.

Keep in mind that the *MAFwApplication* is actually dynamically generating and processing a *processor.ProcessorList* according to the information stored in the `steering file`. The exit code for the whole app is linked to the output of the last executed processor in the list.

The reason for this is clearly explained. We have seen in a previous *section* that the execution of a *processor.ProcessorList* can be modified using the *ProcessorExitStatus*. This parameter can assume three different values: **Successful**, **Failed** and **Aborted**. In MAFw convention, when a processor is aborted, then the execution of the processor list is immediately interrupted, thus resulting in the MAFw executable to return a no zero exit code.

On the contrary, when a processor exit status is Failed, the execution of the processor list is continued until the end of the list or the first processor terminating with an Abort status. The MAFw executable exit code is affected by the the processor exit status of the last executed processor, so it might be zero even if there were several processors with Failed exit status in the middle of the list.

The use of the exit code is particularly relevant if you want to execute MAFw on a headless cluster. You can, for example, trigger retry or logging actions when the execution fails.

7.8 Commands abbreviation

The MAFw executable is based on the very powerful click library and allows the execution of complex nested commands with options and arguments.

You can get the list of available commands (for each level) using the `-h` option and keep in mind that MAFw allows you to shorten the command as long as the command selection will be unique, so actually `mafw list` and `mafw l` are both accepted and produce the same effect.

7.9 Shell completion

MAFw also exposes shell TAB completion through the `completion` command group. The feature is available for `bash`, `zsh` and `fish` only.

Install it in the active virtual environment with:

```
mafw completion install
```

Issuing this command will install the mafw completion in the local virtual environment and will make it available the next time you activate the environment. The installer writes the generated completion script into the virtual environment, updates the activation script, and then asks you to exit and re-enter the environment so the new hook can be loaded automatically.

If you only want to activate directly (but not in a permanent way) the completion functionality, use:

```
eval "$(mafw completion show)"
```

7.10 What's next

You believed we were done? Not really. We have just started, this was just the appetizer! Get ready for the next course, when we will enjoy the power of relational databases!

CHAPTER 8

DATABASE: YOUR NEW BUDDY!

As a scientist you are used to work with many software tools and often you need to write your own programs because the ones available do not really match your needs. Databases are not so common among scientists, we do not really understand why, but nevertheless their strength is beyond question.

The demonstration of the power and usefulness of a database assisted analysis framework will become clear and evident during our *tutorial* where we will build a small analytical experiment from scratch one step at the time.

For the time being, let us concentrate a bit on the technical aspect without delving too deeply into the specifics.

8.1 Database: one name, many implementations

Database is really a generic term and, from a technical point of view, one should try to be more specific and define better what kind of database we are talking about. We are not just referring to the brand or the producing software house: there are indeed different database architectures, but the one best suited for our application is the **relational database**, where each entity can be related directly to one or more other entities. You can read about relational databases on wikipedia for example, and if you find it too complicated, have a look at this shorter and easier version.

The software market is full of relational databases, from the very common MySQL to the very famous ORACLE, passing through the open source PostgreSQL to finish with SQLite the simplest and most used database in the world. As you may have guessed from their names, they are all sharing the same query language (*SQL*: structured query language), making it rather simple to have an abstract interface, that is to say a layer in between the user and the actual database that allows your code to work in the same way irrespectively of the specific implementation.

8.2 Peewee: a simple and small ORM

Of course there are also lots of different abstract interfaces, some more performing than the others. We have selected peewee, because it is lightweight, easy to understand and to use, and it works with several different implementations.

peewee is a ORM (promised this is the last acronym for this page!), it is to say an object relational mappers, or in simpler words a smart way to connect the tables in your database with python classes in your code. Have a look at this interesting article for a more detailed explanation.

peewee offers at least three different backends: SQLite, MySQL and PostgreSQL. If the size of your project is small to medium and the analysis is mainly performed on a single computer, then we recommend SQLite: the entire database will be living in a single file on your disc, eliminating the need for IT experts to set up a database server. If you are aiming for a much bigger project with distributed computing power, then the other two choices are probably equally good and your local IT helpdesk may suggest you what is best for your configuration and the available IT infrastructure. As you see, MAFw is always offering you a tailored solution!

Now, take a short break from this page, move to the peewee documentation and read the *Quickstart* section before coming back here.

8.2.1 Database drivers

If you have installed MAFw via pip without specifying the *all-db* optional features, then your python environment is very likely missing the python drivers to connect to MySQL and PostgreSQL. This is not a bug, but more a feature, because MAFw gives you the freedom to select the database implementation that fits your needs. Sqlite is natively supported by python, so you do not need to install anything extra, but if you want to use MySQL, PostgreSQL or any other DB supported by peewee than it is your responsibility to install in your environment the proper driver. Here you can find a list of DB driver compatible with peewee.

If you want, you can install MAFw adding the *all-db* optional feature and in this way the standard MySQL and PostgreSQL drivers will also be installed.

8.3 One class for each table

As mentioned before, the goal of an ORM is to establish a link between a database table and a python class. You can use the class to retrieve existing rows or to add new ones, and as always, you do not need take care of the boring parts, like establishing the connection, creating the table and so on, because this is the task of MAFw and we do it gladly for you!

Let us have a look together to the following example. We want a processor that lists recursively all files starting from a given directory and adds the filenames and the file hash digests to a table in the database.

Let us start with some imports

```
1  import datetime
2  from pathlib import Path
3
4  from peewee import DateTimeField, IntegerField, TextField
5
6  import mafw.processor
7  from mafw.db.db_configurations import db_scheme, default_conf
8  from mafw.db.db_model import MAFwBaseModel
9  from mafw.decorators import database_required
10 from mafw.tools.file_tools import file_checksum
```

The crucial one is at line 8, where we import *MAFwBaseModel* that is the base model for all the tables we want to handle with MAFw. Your tables **must inherit** from that one, if you want the *Processor* to take care of handling the link between the class and the table in the database. At line 4, we import some classes from peewee, that will define the columns in our model class and consequently in our DB table.

Now let us create the model class for our table of files.

```
1  class File(MAFwBaseModel):
2      """The Model class representing the table in the database"""
3
4      filename = TextField(primary_key=True)
5      digest = TextField()
6      creation_date = DateTimeField()
7      file_size = IntegerField()
```

> **Note**
>
> A much better implementation of a similar class will be given *later on* demonstrating the power of custom defined fields.

As you can see the class definition is extremely simple. We define a class attribute for each of the columns we want to have in the table and we can choose the field type from a long list of available ones or we can even easily implement our owns. The role of a field is to adapt the type of the column from the native python type to the native database type and vice versa.

Our table will have just four columns, but you can have as many as you like. We will have one text field with the full filename, another text containing the hexadecimal hashlib digest of the file, the creation date for which we will use a datetime field, and finally a file_size field of integer type. We will be using the filename column as a primary key, because there cannot be two files with the same filename. On the contrary, there might be two identical files having the same hash but different filenames. According to many good database experts, using a not numeric primary key is not a good practice, but for our small example it is very practical.

If you do not specify any primary key, the ORM will add an additional number auto-incrementing column for this purpose. If you want to specify multiple primary keys, this is what you should do. If you want to create a model without a primary key, here is what you need to do.

The ORM will define the actual name of the table in the database and the column names. You do not need to worry about this!

And now comes the processor that will be doing the real work, it is to say, filling the File table.

```
@database_required
class FillFileTableProcessor(mafw.processor.Processor):
    """Processor to fill a table with the content of a directory"""

    root_folder: Path = mafw.processor.ActiveParameter(
        'root_folder', default=Path.cwd(), help_doc='The root folder for the file␣
↪listing'
    )

    def __init__(self, *args, **kwargs):
        """
        Constructor parameter:

        :param root_folder: ActiveParameter corresponding to the directory from
            where to start the recursive search
        :type root_folder: Path, Optional
        """
        super().__init__(*args, **kwargs)
        self.data: list[dict] = []

    def format_progress_message(self):
        self.progress_message = f'Upserting {self.item.name}'

    def start(self):
        """Starts the execution.

        Be sure that the table corresponding to the File model exists.
        It it does already exists, it is not a problem.
        """
        super().start()
        self.database.create_tables((File,))

    def get_items(self) -> list[Path]:
        """Retrieves the list of files.

        Insert or update the files from the root folder to the database

        :return: The list of full filename
        :rtype: list[Path]
        """
        file_list = []
        if self.root_folder.is_file():
            return [self.root_folder]
```

(continues on next page)

(continued from previous page)

```
43          elif self.root_folder.is_dir():
44              for f in self.root_folder.glob('**/*'):
45                  if f.is_file():
46                      file_list.append(f)
47          else:  # root_file is a glob
48              for f in self.root_folder.parent.glob(self.root_folder.name):
49                  file_list.append(f)
50          return file_list
51
52      def process(self):
53          """Add all information to the data list"""
54          self.data.append(
55              dict(
56                  filename=str(self.item),
57                  digest=file_checksum(self.item),
58                  file_size=self.item.stat().st_size,
59                  creation_date=datetime.datetime.fromtimestamp(self.item.stat().st_
    ↪mtime),
60              )
61          )
62
63      def finish(self):
64          """Transfers all the data to the File table via an atomic transaction."""
65          with self.database.atomic():
66              File.insert_many(self.data).on_conflict_replace().execute()
67          super().finish()
```

The first thing to notice is at line 1, where we used the decorator *database_required()*. The use of this decorator is actually not compulsory, its goal is to raise an exception if the user tries to execute the processor without having a properly initialized database.

At line 30, in the *start* method we ask the database to create the table corresponding to our *File* model. If the table already exists, then nothing will happen.

In the *process* method we will store all the information we have collected from the files into a list and we interact with the database only in the *finish* method. At line 65, we use a context manager to create an atomic transaction and then, at line 66, we insert in the *File* all our entries and in case a row with the same primary key exists, then it is replaced.

We could have used several different insert approaches, here below are few examples:

```
# create an instance of File with all fields initialized
new_file = File(filename=str(self.item),
                digest=file_checksum(self.item),
                file_size=self.item.stat().st_size,
                creation_date=datetime.datetime.fromtimestamp(self.item.stat().st_
    ↪mtime))
new_file.save() # new_file is now stored in the database

# create an instance of File and add the fields later
new_file = File()
new_file.filename = str(self.item)
new_file.digest = file_checksum(self.item)
new_file.file_size = self.item.stat().st_size
new_file.creation_data = datetime.datetime.fromtimestamp(self.item.stat().st_mtime)
new_file.save()

# create and insert directly
```

(continues on next page)

(continued from previous page)

```
new_file = File.create(filename=str(self.item),
                       digest=file_checksum(self.item),
                       file_size=self.item.stat().st_size,
                       creation_date=datetime.datetime.fromtimestamp(self.item.stat().st_
↪mtime))
```

The approach to follow depends on various factor. Keep in mind that peewee operates by default in auto commit mode, meaning that for each database interaction, it creates a transaction to do the operation and it closes afterwards.

To be more performant from the database point of view, especially when you have several operations that can be grouped together, you can create an atomic transaction where the ORM will open one transaction only to perform all the required operations.

What we have done in the *finish* method is actually known as an upsert. It means that we will be inserting new items or updating them if they exist already.

8.4 Ready, go!

We have prepared the code, now we can try to run it. We can do it directly from a script

```
if __name__ == '__main__':
    database_conf = default_conf['sqlite']
    database_conf['URL'] = db_scheme['sqlite'] + str( Path.cwd() / Path('test.db'))

    db_proc = FillFileTableProcessor(root_folder =r'C:\Users\bulghao\Documents\
↪autorad-analysis\EdgeTrimming',
                                     database_conf=database_conf)

    db_proc.execute()
```

or in a more elegant way we can use the mafw app to run, but first we need to generate the proper steering file.

Console

```
c:\> mafw steering db-processor.toml
A generic steering file has been saved in db-processor.toml.
Open it in your favourite text editor, change the processors_to_run list and save it.

To execute it launch: mafw run db-processor.toml.
```

TOML

```
# MAFw steering file generated on 2024-11-24 22:13:38.248423

# uncomment the line below and insert the processors you want to run from the␣
↪available processor list
processors_to_run = ["FillFileTableProcessor"]

# customise the name of the analysis
analysis_name = "mafw analysis"
analysis_description = "Using the DB"
available_processors = ["AccumulatorProcessor", "GaussAdder", "ModifyLoopProcessor",
↪"FillFileTableProcessor", "PrimeFinder"]

[DBConfiguration]
URL = "sqlite:///file-db.db" # Change the protocol depending on the DB type. Update␣
```

(continues on next page)

(continued from previous page)

```
↪this file to the path of your DB.

[DBConfiguration.parameters.sqlite]
# you do not need to add this empty table, but it does not hurt and
# it makes the file easier to read.

[DBConfiguration.parameters.sqlite.pragmas] # Leave these default values, unless you␣
↪know what you are doing!
journal_mode = "wal"
cache_size = -64000
foreign_keys = 1
synchronous = 0

[FillFileTableProcessor] # Processor to fill a table with the content of a directory
root_folder = 'C:\Users\bulghao\PycharmProjects\mafw' # The root folder for the file␣
↪listing

[UserInterface] # Specify UI options
interface = "rich" # Default "rich", backup "console"
```

If you look at the steering file, you will notice that there is a `DBConfiguration` section, where we define the most important variable, it is to say the DB URL. This is not only specifying where the database can be found, but also the actual implementation of the database. In this case, it will be a sqlite database located in the file `file-db.db` inside the current directory.

There is also an additional sub table, named `parameters.sqlite.pragmas`, containing advanced options for the sqlite DB. Unless you really know what you are doing, otherwise, you should leave them as they are.

In the following *Configuring other types of databases*, we will cover the case you want to use another DB implementation different from SQLite.

In the `FillFileTableProcessor` you can find the standard configuration of its processor parameters.

Now we are really ready to run our first DB processor and with a bit of luck, you should get your DB created and filled.

How to check the content of a DB?

There are several tools serving this purpose. One of those is dbeaver that works with all kind of databases offering an open source community version that you can download and install.

8.4.1 Configuring other types of databases

In the previous example, we have seen how to configure a simple SQLite database. For this database, you just need to indicate in the URL field the path on the local disc where the database file is stored.

SQLite does not require any user name nor password and there are no other fields to be provided. Nevertheless, it is worth adding the previously mentioned `parameters.sqlite.pragmas` section to assure the best functionality of peewee.

In the case of MySQL and PostgreSQL, the URL should point to the server where the database is running. This could be the localhost but also any other network destination. Along with the server destination, you need also to specify the port and the database name; authentication is now provided through a dedicated `DBConfiguration.authentication` subtable so transport, authentication, and backend-specific parameters stay separate.

The authentication subtable supports three methods:

- **env**: the `username` and `password` entries are the *environment variable names* to resolve at runtime. This option is very convenient for remote execution on a batch system because you can easily and securely define your connection credentials in environmental variables.

- **inline**: the `username` and `password` values are stored directly in the steering file (not recommended). Use this with extreme care, because you risk to leak your credential by commiting the steering file to the git repository.
- **file**: the `passfile` entry points to a password file. For MySQL this maps to `read_default_file`; for PostgreSQL this maps to `passfile` (and omitting it lets libpq use the default `.pgpass`). The file authentication is the most suited for local execution, because you just edit the password file according to the DB requirements and you are good to go without having to export environmental variables.

Have a look at the exemplary steering files with the corresponding configurations here below.

SQLite

```
[DBConfiguration]
URL = "sqlite:///file-db.db" # change the filename to the absolute path of the db

[DBConfiguration.parameters.sqlite]

[DBConfiguration.parameters.sqlite.pragmas] # Leave these default values, unless you␣
↪know what you are doing!
journal_mode = "wal"
cache_size = -64000
foreign_keys = 1
synchronous = 0
```

PostgreSQL

```
[DBConfiguration]
# Update the database server and the database name to reflect your configuration
URL = "postgresql://database-server:5432/database-name"

[DBConfiguration.authentication]
method = "file"
username = "username"
passfile = "~/.pgpass"

[DBConfiguration.parameters.postgresql]
```

Instruction on how to create a PostgreSQL password file are provided here. This is an example:

```
database-server:5432:database-name:username:password
```

MySQL

```
[DBConfiguration]
# Update the database server and the database name to reflect your configuration
URL = "mysql://database-server:3306/database-name"

[DBConfiguration.authentication]
method = "file"
username = "username"
passfile = "~/.my.cnf"

[DBConfiguration.parameters.mysql]
```

Instruction on how to create a MySQL password file are provided here. This is an example:

```
[client]
user=username
password=password
host=database-server
```

Changing the DB URL

Instead of hard coding the database URL, you can use the environment variable expansion supported by MAFw steering files. This lets you keep one steering file and switch between local, testing, or production databases by exporting a different variable before running the analysis.

8.5 The advantages of using MAFwBaseModel

In the previous code snippets we have implemented our Model classes as sub-class of *MAFwBaseModel*. This is not just a detail because *MAFwBaseModel* is offering some embedded advantages compared to the base model class of peewee. In the following subsections we will explicitly describe them all. For additional details, you can also read the API documentation of *MAFwBaseModel* and of *RegisteredMeta* the meta class where all this magic is taking place.

8.5.1 Automatic join helper

MAFwBaseModel exposes *join_dependent_models()*, which returns a `Select` statement that already joins every foreign-key dependency declared on the model. Each related row is attached to the resulting instances via an attribute named `_<field name>`, mirroring the naming convention used in manual `join_from` chains (e.g. `RawImage._grid`). Processors can call this helper to avoid writing long `join` cascades and focus only on the filters coming from the steering file and any additional domain-specific conditions.

Example:

```
query = RawImage.join_dependent_models().where(RawImage.grid_id != 0)
query = query.where(self.filter_register.filter_all())
```

This utility is especially helpful when the input model links to multiple tables, because it automatically generates the dependencies' `JOIN` clauses without forcing you to repeat the same boilerplate every time.

8.5.2 Automatic model registration

We will explain you the relevance of this point with an example.

Imagine that you have a processor that is performing some calculations; the processor can operate on real data or on simulated data following the same workflow, the only difference is that we will be taking input items from two different tables, one for real data and one for simulated data. Similarly we would like to save the output into two different tables.

The most convenient approach would be to set the input and output Models as processor parameters, but a Model is a python class and we could not provide this via our steering file. There we can imagine to provide a string representing the name of the corresponding DB table, but peewee is missing the capability to retrieve the Model class given the name of the corresponding DB table or the model name, the last possibility would be to build up a custom selection logic based on *if/else* blocks.

The automatic registration and the *mafw_model_register* are there exactly to fill this hole. When you define a new model class inheriting from *MAFwBaseModel*, this will be automatically added to model register and then you will be able to retrieve it using the *ModelRegister.get_model()* method[1].

Here is a little example to make it simpler.

[1] The automatic registration of a class at his definition is a nice piece of syntactic sugar obtained via the use metaclasses. See the *API* for more.

db_model.py

```
# file: db_model.py
# this is the module where all our models are defined

from peewee import IntegerField
from mafw.db_model import MAFwBaseModel

class MyRegisteredModel(MAFwBaseModel):
    integer_num = IntegerField(primary_key=True)
```

my_processor.py

```
# file: my_processor.py
# this it the module where the processors are defined

from mafw.processor import Processor
from mafw.db.db_model import mafw_model_register

import db_model.py

class MyProcessor(Processor):
    # put here all your definitions...
    def get_items(self):
        # we want to process all items in the input model.
        input_model = mafw_model_register.get_model('my_registered_model')
        return input_model.select()
```

In the my_processor.py file you can see how to retrieve a Model class from the register. You can use either the full table name or if you wish the Model name (MyRegisteredModel) as a key. It is important to notice line 7 where we import db_model with all our model definitions. This is necessary because the automatic registration occurs when the interpreter is processing the definition of a class inheriting from *MAFwBaseModel*. The imported module is actually not used in the processor, but we need to be sure that the interpreter has read the definition of the model. In a *following section*, you will see a smart method to avoid this useless import.

Exclude from automatic registration

There might be cases, not many to be honest, in which you want a Model class not to be registered. A typical example is the case of a Model class used in your project as a base for all your models. It makes no sense to have this *abstract* model in the registry because you will never use it.

If you want to prevent the automatic registration then you simply have to specify it in the model definition as in the following example:

```
from mafw.db.db_model import MAFwBaseModel

class MyBaseModel(MAFwBaseModel, do_not_register=True):
    pass

class MyConcreteModel(MyBaseModel):
    pass
```

Have a look at the definition of MyBaseModel. Along with the specification of the base class, we provided as well the extra keyword argument *do_not_register*. In this example, MyBaseModel will not be registered in the *mafw_model_register*, while MyConcreteModel is.

Automatic import of db modules

Having the automatic registration is for sure a big step forward, something less for you to take care, but having to import the module with the model class definition is a bit annoying, in particular if your code is using a linter where unused_import (F401) are often automatically removed.

The idea is to use once again the plugin mechanism of MAFw to inform the runner (`mafw`), that the module including all the db models should be imported. This is how you can achieve this:

The *register_db_model_modules* is expected to return a list of module fully qualified module names. Those strings will be then automatically passed to the importlib.import_module function in order to be loaded. It is important that the string is properly formatted and the module is correctly spelled, otherwise MAFw will raise an ImportError.

8.5.3 Trigger support

The use of triggers and their advantages for an efficient and reliable data analysis will be discussed at length in the *following chapter*. Some special triggers can also be automatically generated as explained *here*.

8.5.4 Customizable table name

The default naming convention adopted by peewee is to transform the CamelCase model name is a snake_case table name. Unless you want to access the tables using low level SQL statements or from an external tool like dbeaver, then knowing the actual table name is not very important.

Peewee offers the possibility to customize the table names and MAFwBaseModel expand this even further, with the possibility to add a prefix, a suffix or both to the camel_case table name. This feature is very handy when you are handling a large database with several tens of tables. When browsing the database index from dbeaver, tables are normally sorted in alphabetical order and thus finding a specific table can be frustrating. You may want to prefix all the tables belonging to a specific part of your experiment with a given word, in order to have them all grouped together[2].

This is how:

```
from mafw.db.db_model import MAFwBaseModel

class SimulationModel(MAFwBaseModel, do_not_register=True):
    class Meta:
        prefix = 'simul'

class SimulationResult(SimulationModel):
    # put here your fields
    pass

class DataModel(MAFwBaseModel, do_not_register=True):
    class Meta:
        prefix = 'data'

class DataResult(DataModel):
    # put here your fields
    pass
```

In this example we have defined two base models (that are not registered automatically), one for simulation and one for real data results. In the Meta class definition of the two base models we have assigned a value to the attribute `prefix`. This attribute will be inherited by all their subclasses, so that the table name corresponding to SimulationResult will be *simul_simulation_result*. A similar effect can be obtained setting the `suffix` parameter.

[2] MySQL does not directly support adding WHEN conditions to the trigger, but a similar behaviour is obtainable using an IF statement in the trigger SQL body. This adaptation is automatically implemented by the *MySQLDialect*.

See also

See also *make_prefixed_suffixed_name()* for more details on the naming convention.

8.5.5 Automatic table creation

In our first *processor using the database*, we have created the table corresponding to the File model manually in the start method. This operation must be done before we try to access the content of the table, otherwise the database will generate an error. If a table is already existing, then the create_table command is actually ignored.

One of the biggest advantage of inheriting from *MAFwBaseModel* is that the creation of all models can be automatized by using a processor included in MAFw library: the *TableCreator*. You can simply include this processor in your TOML steering file, possibly as a first processor in the list and afterwards you can forget about creating tables.

This processor is customizable via two parameters:

1. *apply_only_to_prefix*: allows to restrict the creation of tables whose name is starting with a given prefix. More than one prefix can be provided.
2. *force_recreate*: will force the tables to be recreated in the case they are already existing. This is rather **dangerous** because, the processor will actually drop all tables before creating them causing the loss of all your data. Use it with extreme care!!!

You still have the freedom to disable this automatic feature by setting the *automatic_creation* flag in the Meta class to False, like in this example:

```
from mafw.db.db_model import MAFwBaseModel

class NoAutoModel(MAFwBaseModel):
    # put here your fields
    class Meta:
        automatic_creation = False
```

Remember: only registered models can have tables automatically created. If you decided not to register a model, then you will have to manually create the corresponding table before using it.

8.6 Triggers: when the database works on its own

In the next paragraphs we will spend a few minutes understanding the roles of Triggers. Those are database entities performing some actions in response of specific events. You can have, for example, a trigger that is inserting a row in TableB whenever a row is inserted in TableA. If you are not really familiar with triggers, this is a brief introduction.

Triggers are very handy for many applications, and in our *tutorial* we will see an interesting case, but they tend to struggle with ORM in general. In fact, no ORM system is natively supporting triggers. The reason is very simple. In an ORM, the application (the python code, if you wish) is the main actor and the database is just playing the role of the passive buddy. From the point of view of an ORM based application, if you want to have a trigger, then just write the needed lines of python code to have the actions performed in the other tables. It makes totally sense, you have only one active player and it simplifies the debugging because if something is going wrong, it can only be a fault of the application.

The standard implementation of trigger-like functions with ORM is to use signals, where you can have callbacks called before and after high level ORM APIs calls to the underlying database. Signals are good, but they are not free from disadvantages: at a first glance, they look like a neat solution, but as soon as the number of callbacks is growing, it may become difficult to follow a linear path in the application debugging. Second, if you do a change in the database from another application, like the dbeaver browser, then none of your codified triggers will be executed. Moreover in the case of Peewee, signals work only on Model instances, so all bulk inserts and updates are excluded.

Having triggers in the database would assure that irrespectively of the source of the change, they will always be executed, but as mentioned above, the user will have to be more careful in the debug because also the database is now playing an active role.

We let you decide what is the best solution. If you want to follow the pure ORM approach, then all models inheriting from *MAFwBaseModel* have the possibility to use signals. If you want to have triggers, you can also do so. An example for both approaches is shown here below.

8.6.1 The signal approach

As mentioned above, the signal approach is the favourite one if you plan to make all changes to the database only via your python code. If you are considering making changes also from other applications, then you should better use the trigger approach.

Another limitation is that only model instances emit signals. Everytime you use a *classmethod* of a Model, then no signals will be emitted.

The signal dispatching pattern functionality is achieved by linking the signal emitted by a sender in some specific circumstances to a handler that is receiving this signal and performing some additional operations (not necessarily database operations).

Every model class has five different signals:

1. **pre_save**: emitted just before that a model instance is saved;
2. **post_save**: emitted just after the saving of a model instance in the DB;
3. **pre_delete**: emitted just before deleting a model instance in the DB;
4. **post_delete**: emitted just after deleting a model instance from the DB;
5. **pre_init**: emitted just after the init method of the class is invoked. Note that the *pre* is actually a *post* in the case of init.

Let us try to understand how this works with the next example.

Listing 8.1: A test with signal

```
class MyTable(MAFwBaseModel):
    id_ = AutoField(primary_key=True)
    integer = IntegerField()
    float_num = FloatField()

class TargetTable(MAFwBaseModel):
    id_ = ForeignKeyField(MyTable, on_delete='cascade', primary_key=True, backref=
↪'half')
    another_float_num = FloatField()

    @post_save(sender=MyTable, name='my_table_after_save_handler')
    def post_save_of_my_table(sender: type(MAFwBaseModel), instance: MAFwBaseModel,␣
↪created: bool):
        """
        Handler for the post save signal.

        The post_save decorator is taking care of making the connection.
        The sender specified in the decorator argument is assuring that only signals␣
↪generated from MyClass will be
        dispatched to this handler.

        The name in the decorator is optional and can be use if we want to disconnect␣
↪the signal from the handler.

        :param sender: The Model class sending this signal.
```

(continues on next page)

(continued from previous page)

```
22          :type sender: type(Model)
23          :param instance: The actual instance sending the signal.
24          :type instance: Model
25          :param created: Bool flag if the instance has been created.
26          :type created: bool
27          """
28          TargetTable.insert({'id__id': instance.id, 'another_float_num': instance.
    ↪float_num / 2}).execute()
29
30  database: Database = SqliteDatabase(':memory:', pragmas=DEFAULT_SQLITE_PRAGMAS)
31  database.connect()
32  database_proxy.initialize(database)
33  database.create_tables([MyTable, TargetTable], safe=True)
34
35  MyTable.delete().execute()
36  TargetTable.delete().execute()
37
38  # insert a single row in MyTable with the save method.
39  my_table = MyTable()
40  my_table.integer = 20
41  my_table.float_num = 32.16
42  my_table.save()
43  # after the save query is done, the signal mechanism will call the
44  # post_save_trigger_of_my_table and perform an insert on the target
45  # table as well.
46  assert MyTable.select().count() == 1
47  assert TargetTable.select().count() == 1
48
49  # add some bulk data to MyTable
50  data = []
51  for i in range(100):
52      data.append(dict(integer=random.randint(i, 10 * i), float_num=random.gauss(i, 2 *␣
    ↪i)))
53  MyTable.insert_many(data).execute()
54  # this is done via the Model class and not via a concrete instance of the Model, so␣
    ↪no signals will be emitted.
55
56  assert MyTable.select().count() == 101
57  assert TargetTable.select().count() == 1
```

We created two tables linked via a foreign key. The goal is that everytime we fill in a row in `MyTable`, a row is also added to TargetTable with the same id but where the value of another_float_num is just half of the original float_num. The example is stupid, but it is good enough for our demonstration.

The signal part is coded in the lines from 11 to 28 (mainly doc strings). We use the `post_save` decorator to connect MyTable to the `post_save_of_my_table` function where an insert in the TargetTable will be made.

The code is rather simple to follow. Just to be sure, we empty the two tables, then we create an instance of the MyTable model, to set the integer and the float_num column. When we save the new row, the post_save signal of MyTable is emitted and the handler is reacting by creating an entry in the TargetTable as well. In fact the number of rows of both tables are equal to 1.

What happens later is to demonstrate the weak point of signals. At line 53, we insert several rows via a `insert_many`. It must be noted that the insert_many is a classmethod applied directly to the model class. The consequence is that the signal handler will not be invoked and no extra rows will be added to the TargetTable.

8.6.2 The trigger approach

In order to use a trigger you need to create it. This is an entity that lives in the database, so you would need the database itself to create it.

MAFw is providing a *Trigger* class that helps you in creating the required SQL query that needed to be issued in order to create the trigger. Once it is created it will operate continuously.

If you have a look at the CREATE TRIGGER SQL command you will see that it starts with the definition of when the trigger is entering into play (BEFORE/AFTER) and which operation (INSERT/DELETE/UPDATE) of which table. Then there is a section enclosed by the BEGIN and END keywords, where you can have as many SQL queries as you like.

The same structure is reproduced in the *Trigger* class. In the constructor, we will pass the arguments related to the configuration of the trigger itself. Then you can add as many SQL statement as you wish.

Python

Listing 8.2: python Trigger class

```python
from mafw.db.db_model import Trigger

new_trigger = Trigger('trigger_after_update', (TriggerWhen.After,
        TriggerAction.Update), 'my_table', safe=False, for_each_row=False)
new_trigger.add_sql('INSERT INTO another_table (col1, col2) VALUES (1, 2)')
new_trigger.add_sql('INSERT INTO third_table (col1, col2) VALUES (2, 3)'))
new_trigger.create()
```

SQL

Listing 8.3: emitted SQL

```sql
CREATE TRIGGER trigger_after_update
AFTER UPDATE  ON my_table

BEGIN
    INSERT INTO another_table (col1, col2) VALUES (1, 2);
    INSERT INTO third_table (col1, col2) VALUES (2, 3);
END;
```

Now let us have a look at how you can use this, following one of our test benches.

Standalone triggers

Listing 8.4: A test Trigger created manually

```python
def test_manually_created_trigger():
    class MyTable(MAFwBaseModel):
        id_ = AutoField(primary_key=True)
        integer = IntegerField()
        float_num = FloatField()

    class TargetTable(MAFwBaseModel):
        id_ = ForeignKeyField(MyTable, on_delete='cascade', primary_key=True, backref=
↪'half')
        half_float_num = FloatField()

    database: Database = SqliteDatabase(':memory:', pragmas=DEFAULT_SQLITE_PRAGMAS)
    database.connect()
```

(continues on next page)

(continued from previous page)

```
    database_proxy.initialize(database)
    database.create_tables([MyTable, TargetTable], safe=True)

    MyTable.delete().execute()
    TargetTable.delete().execute()

    # manually create a trigger
    trig = Trigger('mytable_after_insert', (TriggerWhen.After, TriggerAction.Insert), MyTable, safe=True)
    trig.add_sql('INSERT INTO target_table (id__id, half_float_num) VALUES (NEW.id_, NEW.float_num / 2)')
    database.execute_sql(trig.create())

    # add some data for testing to the first table
    data = []
    for i in range(100):
        data.append(dict(integer=random.randint(i, 10 * i), float_num=random.gauss(i, 2 * i)))
    MyTable.insert_many(data).execute()

    # check that the target table got the right entries
    for row in MyTable.select(MyTable.float_num, TargetTable.half_float_num).join(TargetTable).namedtuples():
        assert row.float_num == 2 * row.half_float_num

    assert MyTable.select().count() == TargetTable.select().count()
```

In lines 20 - 22, we create a trigger and we ask the database to execute the generated SQL statement.

We insert 100 rows using the insert many class method and the trigger is doing its job in the background filling the other table. We can check that the values in the two tables are matching our expectations.

The drawback of this approach is that you may have triggers created all around your code, making your code a bit messy.

Model embedded triggers

An alternative approach is to define the trigger within the Model class, allowing it to be created simultaneously with model table. This is demonstrated in the code example below.

Listing 8.5: A test Trigger created within the Model

```
# the trigger is directly defined in the class.
class MyTable(MAFwBaseModel):
    id_ = AutoField(primary_key=True)
    integer = IntegerField()
    float_num = FloatField()

    @classmethod
    def triggers(cls):
        return [
            Trigger('mytable_after_insert', (TriggerWhen.After, TriggerAction.Insert), cls, safe=True).add_sql(
                'INSERT INTO target_table (id__id, half_float_num) VALUES (NEW.id_, NEW.float_num / 2)'
            )
        ]

```

(continues on next page)

(continued from previous page)

```
15 class TargetTable(MAFwBaseModel):
16     id_ = ForeignKeyField(MyTable, on_delete='cascade', primary_key=True, backref=
   ↪'half')
17     half_float_num = FloatField()
18
19 database: Database = SqliteDatabase(':memory:', pragmas=DEFAULT_SQLITE_PRAGMAS)
20 database.connect()
21 database_proxy.initialize(database)
22 database.create_tables([MyTable, TargetTable], safe=True)
```

This approach is much cleaner. The Trigger is stored directly in the Model (lines 8 - 13). In the specific case, the triggers method returned one trigger only, but you can return as many as you like. When the tables are created (line 22), all the triggers will also be created.

In the example above, you have written the SQL statement directly, but nobody is preventing you to use peewee queries for this purpose. See below, how exactly the same trigger might be re-written, using an insert statement:

Listing 8.6: A test Trigger created within the Model using an Insert statement

```
1  class MyTable(MAFwBaseModel):
2      id_ = AutoField(primary_key=True)
3      integer = IntegerField()
4      float_num = FloatField()
5
6      @classmethod
7      def triggers(cls):
8          trigger = Trigger('mytable_after_insert', (TriggerWhen.After, TriggerAction.
   ↪Insert), cls, safe=True)
9          sql = TargetTable.insert(id_=SQL('NEW.id_'), half_float_num=SQL('NEW.float_
   ↪num/2'))
10         trigger.add_sql(sql)
11         return [trigger]
12
13     class Meta:
14         depends_on = [TargetTable]
```

The key point here is at line 9, where the actual insert statement is generated by peewee (just for your information, you have generated the statement, but you have not *execute it*) and added to the existing trigger.

In the last two highlighted lines, we are overloading the Meta class, specifying that MyTable, depends on TargetTable, so that when the create_tables is issued, they are built in the right order. This is not necessary if you follow the previous approach because the trigger will be very likely executed only after that the tables have been created.

Warning

Even though starting from MAFw release v1.1.0, triggers are now properly generated for the three main *database backends*, its use has been deeply tested only with SQLite. For this reason, we (MAFw developers) encourage the user community to work also with other DBs and, in case, submit bugs or feature request.

Disabling triggers

Not all database implementations provide the same option to temporarily disable one or more triggers. In order to cope with this limitation, MAFw is providing a general solution that is always working independently of the concrete implementation of the database.

The standard SQL trigger definition allows to have one or more WHEN clauses[3], meaning that the firing of a trigger

[3] In some databases, like mysql, one can use the *schema* to group tables together.

script might be limited to the case in which some other external conditions are met.

In order to achieve that, we use one of our *standard tables*, that are automatically created in every MAFw database.

This is the TriggerStatus table as you can see it in the snippet below:

Listing 8.7: TriggerStatus model

```
class TriggerStatus(StandardTable):
    """A Model for the trigger status"""

    trigger_type_id = AutoField(primary_key=True, help_text='Primary key')
    trigger_type = TextField(
        help_text='You can use it to specify the type (DELETE/INSERT/UPDATE) or the␣
↪name of a specific trigger'
    )
    status = BooleanField(default=True, help_text='False (0) = disable / True (1) =␣
↪enable')

    # noinspection PyProtectedMember
    @classmethod
    def init(cls) -> None:
        """Resets all triggers to enable when the database connection is opened."""
        data = [
            dict(trigger_type_id=1, trigger_type='DELETE', status=True),
            dict(trigger_type_id=2, trigger_type='INSERT', status=True),
            dict(trigger_type_id=3, trigger_type='UPDATE', status=True),
            dict(trigger_type_id=4, trigger_type='DELETE_FILES', status=True),
        ]

        # this is used just to make mypy happy
        # cls and meta_cls are exactly the same thing
        meta_cls = cast(PeeweeModelWithMeta, cls)

        db_proxy = meta_cls._meta.database
        if isinstance(db_proxy, peewee.DatabaseProxy):
            db = cast(peewee.Database, db_proxy.obj)
        else:
            db = cast(peewee.Database, db_proxy)

        if isinstance(db, peewee.PostgresqlDatabase):
            cls.insert_many(data).on_conflict(
                'update', conflict_target=[cls.trigger_type_id], update={cls.status:␣
↪True}
            ).execute()
        else:
            cls.insert_many(data).on_conflict_replace().execute()
```

You can use the `trigger_type` column to specify a generic family of triggers (DELETE/INSERT/UPDATE) or the name of a specific trigger. By default a trigger is active (status = 1), but you can easily disable it by changing its status to 0.

To use this functionality, the Trigger definition should include a WHEN clause as described in this modified model definition.

Listing 8.8: Trigger definition with when conditions.

```
class MyTable(MAFwBaseModel):
    id_ = AutoField(primary_key=True)
    integer = IntegerField()
```

(continues on next page)

(continued from previous page)

```
    float_num = FloatField()

    @classmethod
    def triggers(cls):
        return [
            Trigger('mytable_after_insert', (TriggerWhen.After, TriggerAction.Insert),
↪ cls, safe=True)
            .add_sql('INSERT INTO target_table (id__id, half_float_num) VALUES (NEW.
↪id_, NEW.float_num / 2)')
            .add_when('1 == (SELECT status FROM trigger_status WHERE trigger_type ==
↪"INSERT")')
        ]
```

To facilitate the temporary disabling of a specific trigger family, MAFw provides a special class *TriggerDisabler* that can be easily used as a context manager in your code. This is an ultra simplified snippet.

Listing 8.9: Use of a context manager to disable a trigger

```
with TriggerDisabler(trigger_type_id = 1):
    # do something without triggering the execution of any trigger of type 1
    # in case of exceptions thrown within the block, the context manager is restoring
    # the trigger status to 1.
```

8.6.3 Triggers on different databases

We have seen that Peewee provides an abstract interface that allows interaction with various SQL databases like MySQL, PostgreSQL, and SQLite.

This abstraction simplifies database operations by enabling the same codebase to work across different database backends, thanks to the common SQL language they all support. However, while these databases share SQL as their query language, they differ in how they handle certain features, such as triggers. Each database has its own peculiarities and syntax for defining and managing triggers, which can lead to inconsistencies when using a single approach across all databases.

To address this challenge, the MAFw introduced the *TriggerDialect* abstract class and three specific implementations for the main databases. Relying on the use of the TriggerDialect class, a syntactically correct SQL statement for the creation or removal of triggers is generated. But, MAFw cannot read the mind of the user (yet!) and given the very different behaviour of the databases, the operation of the triggers will be different.

Have a look at the table below for an illustrative comparison on how triggers are handled by the different databases.

Feature	MySQL	PostgreSQL	SQLite
Trigger Event	• INSERT • UPDATE • DELETE	• INSERT • UPDATE • DELETE • TRUNCATE	• INSERT • UPDATE • DELETE
Trigger Time	• BEFORE • AFTER	• BEFORE • AFTER • INSTEAD OF	• BEFORE • AFTER • INSTEAD OF
Activation	Row-level only	Row-level and statement-level	Row-level and statement-level
Implementation	BEGIN-END block with SQL statements (supports non-standard SQL like SET statements)	Functions written in PL/pgSQL, PL/Perl, PL/Python, etc.	BEGIN-END block with SQL statements
Trigger Per Event	Multiple triggers allowed ordered by creation time	Multiple triggers allowed ordered alphabetically by default, can be specified	Multiple triggers allowed but unspecified execution order
Privileges required	TRIGGER privilege on the table and SUPER or SYSTEM_VARIABLES_ADMIN for DEFINER	CREATE TRIGGER privilege on schema and TRIGGER privilege on table	No specific privilege model
Foreign Key Cascading	Cascaded foreign key actions do not activate triggers	Triggers are activated by cascaded foreign key actions	Triggers are activated by cascaded foreign key actions
Disabled/Enabled Trigger	Yes, using ALTER TABLE ... DISABLE/ENABLE TRIGGER	Yes, using ALTER TABLE ... DISABLE/ENABLE TRIGGER	No direct mechanism to disable

PostgreSQL offers the most comprehensive trigger functionality, with built-in support for both row-level and statement-level triggers, INSTEAD OF triggers for views, and the widest range of programming languages for implementation. Its trigger functions can be written in any supported procedural language, providing considerable flexibility.

MySQL implements triggers using SQL statements within BEGIN-END blocks and only supports row-level triggers. It allows non-standard SQL statements like SET within trigger bodies, making it somewhat more flexible for certain operations. A critical limitation is that MySQL triggers are not activated by cascaded foreign key actions, unlike the other databases. This is a strong limiting factor and the user should consider it when designing the database model to store their data. In this case, it might be convenient to not rely at all on the cascading operations, but to have dedicated triggers for this purpose.

SQLite provides more trigger capabilities than it might initially appear. While its implementation is simpler than PostgreSQL's, it supports both row-level and statement-level triggers (statement-level being the default if FOR EACH ROW is not specified). Like PostgreSQL, SQLite triggers are activated by cascaded foreign key actions, which creates an important behavioral difference compared to MySQL.

When designing database applications that may need to work across different database systems, these implementation differences can lead to subtle bugs, especially around foreign key cascading behavior. MySQL applications that rely on triggers not firing during cascaded operations might behave differently when migrated to PostgreSQL or SQLite. Similarly, applications that depend on statement-level triggers will need to be redesigned when moving from PostgreSQL or SQLite to MySQL.

All so said, even though MAFw provides a way to handle triggers creation and removal in the same way across all the databases, the user who wants to move from one DB implementation to the other should carefully review the content of the trigger body to ensure that the resulting behavior is what is expected.

8.6.4 Debugging triggers

The challenges connected to debugging triggers have been already mentioned several times. It is a block of code that is executed outside the application in full autonomy and you cannot put a breakpoint in the database. If you see that your code is not behaving as expected and you doubt that triggers can be behind the malfunctioning, then the general recommendation is to proceed one step at the time, trying to simplify as much as possible the trigger function.

In all these cases, you will need to drop the triggers from the database and recreate them with the simplified / corrected implementation. This is a bit annoying because it cannot be done directly from the application because very likely you have *embedded* your triggers in the target class, so you have no way to retrieve them.

The solution is to use the `TriggerRefresher` processor. It will take care of dropping all triggers in the database and recreate them from the corresponding model definition. The idea of dropping something is generally rather scary, because it is a undoable operation; but if you put all your trigger definitions inside the various models and they are all subclasses of `MAFwBaseModel`, then they will all recreated using their latest definition.

You can even leave the `TriggerRefresher` processor in your analysis pipelines all the times!

8.7 Standard tables

In the previous section, we discussed a workaround implemented by MAFw to address the limitations of database backends that cannot temporarily disable trigger execution. This is achieved querying a table where the status of a specific trigger or a family of triggers can be toggled from active to inactive and vice-versa.

This `TriggerStatus` model is one of the so-called MAFw standard tables,

The main differences between a Model inheriting from `MAFwBaseModel` and one inheriting from `StandardTable` are two:

1. A standard table has an additional `initialisation method`, that can be used to set or restore the default content of the table.
2. Automatic creation and initialisation performed by MAFw when the database connection is established. In other words, as soon as your processor connect to a database, it will retrieve from the `ModelRegister` all the standard tables, create them and initialise them.

> **Note**
>
> To avoid useless multiple creation and initialisation of standard tables, only the first `Processor` or `processor.ProcessorList` that is establishing a connection to the database, is taking the responsibility to proceed with the standard table creation and initialisation task.
>
> This means that if you manually connect to a database and you pass the database object to your :class:.processor.Processor`/`processor.ProcessorList`, then the standard table creation and initialisation will be skipped, because your `processor.Processor`/`processor.ProcessorList` will think that it was already done when the database connection was established.
>
> The automatic creation and initialisation can be disabled, either using the `processor.Processor.create_standard_tables` argument in the `processor.Processor` or `processor.ProcessorList` constructor, or, if you run the pipelines via the mafw executables using the top level *create_standard_table* variable in your steering file.

The role of those tables is to support the functionality of the whole analysis pipeline, they are rarely the direct input / output of a specific processor. If you want to add your own, just create a model inheriting from `StandardTable` or make your own subclass, to customize for example the prefix, and the automatic registration in the model register will do the trick.

8.7.1 Default standard tables

Along with the TriggerStatus, there are two other relevant standard tables: the *OrphanFile* and the *PlotterOutput*.

OrphanFile: the house of files without a row

This table can be profitably used in conjunction with Triggers. The user can define a trigger fired when a row in a table is deleted. The trigger will then insert all file references contained in the deleted row into the OrphanFile model.

The next time a processor (it does not matter which one) having access to the database is executed, it will query the full list of files from the *OrphanFile* and remove them.

This procedure is needed to avoid having files on your disc without a reference in the database. It is kind of a complementary cleaning up with respect to *another function* you will discover in a moment.

Additional details about this function are provided directly in the *API*.

PlotterOutput: where all figures go.

Plotters are special *Processor* subclasses with the goal of generating a graphical representation of some data you have produced in a previous step.

The output of a plotter is in many cases one or more figure files and instead of having to define a specific table to store just one line, MAFw is providing a common table for all plotters where they can store the reference to their output files linked to the plotter name.

It is very useful because it allows the user to skip the execution of a plotter if its output file already exists on disc.

Triggers are again very relevant, because when a change is made in the data used to generate a plotter output, then the corresponding rows in this table should also be removed, in order to force the regeneration of the output figures with the updated data.

8.8 Custom fields

We have seen in a previous section that there are plenty of field types for you to build up your model classes and that it is also possible to add additional ones. We have made a few for you that are very useful from the point of view of MAFw. The full list is available *here*.

The role of the database in MAFw is to support the input / output operation. You do not need to worry about specifying filenames or paths. Simply instruct the database to retrieve a list of items, and it will automatically provide the various processors with the necessary file paths for analysis.

With this in mind, we have created a *FileNameField*, that is the evolution of a text field accepting a Path object as a python type and converting it into a string for database storage. On top of *FileNameField*, we have made *FileNameListField* that can contain a list of filenames. This second one is more appropriate when your processor is generating a group of files as output. The filenames are stored in the database as a ';' separated string, and they are seen by the python application as a list of Path objects.

Similarly, we have also a *FileChecksumField* to store the string of hexadecimal characters corresponding to the checksum of a file (or a list of files). From the python side, you can assign either the checksum directly, as generated for example by *file_checksum()* or the path to the file, and the field will calculate the checksum automatically.

The *FileNameField* and *FileNameListField* accept an additional argument in their constructor, called `checksum_field`. If you set it to the name of a *FileChecksumField* in the same table, then you do not even have to set the value of the checksum field because this will be automatically calculated when the row is saved.

With these custom fields in mind, our initial definition of a *File table*, can be re-factored as follows:

```
1 from peewee import AutoField
2
3 from mafw.db.db_model import MAFwBaseModel
```

(continues on next page)

(continued from previous page)

```
from mafw.db.fields import FileNameField, FileChecksumField

class File(MAFwBaseModel):
    file_id = AutoField(primary_key=True, help_text='The primary key')
    file_name = FileNameField(checksum_field='file_digest', help_text='The full␣
↪filename')
    file_digest = FileChecksumField(help_text='The hex digest of the file')
```

Pay attention at the definition of the file_name field. The FileNameField constructor takes an optional parameter `checksum_field` that is actually pointing to the variable of the FileChecksumField.

You can use the two custom fields as normal, for example you can do:

```
new_file = File()
# you can assign a Path object.
new_file.file_name = Path('/path/to/some/file')
# the real checksum will be calculated automatically.
# this next line is totally optional, you can leave it out and it will work in the␣
↪same way.
new_file.file_digest = Path('/path/to/some/file')
```

The super power of these two custom fields is that you can remove useless rows from the database, just issuing one command.

8.8.1 Removing widow rows

Due to its I/O support, the database content should always remain aligned with the files on your disc. If you have a row in your database pointing to a missing file, this may cause troubles, because sooner or later, you will try to access this missing file causing an application crash.

In MAFw nomenclature, those rows are called *widows*, following a similar concept in typesetting, because they are a fully valid database entry, but their data counter part on disc disappeared.

To avoid any problem with widow rows, MAFw is supplying a *function* that the processor can invoke in the start method on the Model classes used as input:

```
class MyProcessor(Processor):

    def start():
        super().start()
        remove_widow_db_rows(InputData)
```

The *remove_widow_db_rows()* will check that all the *FileNameField* fields in the table are pointing to existing files on disc. If not, then the row is removed from the database.

The function is not automatically called by any of the Processor super methods. It is up to the user to decide if and when to use it. Its recommended use is in the overload of the *start()* method or as a first action in the *get_items()* in the case of a *for loop* workflow, so that you are sure to re-generate the rows that have been removed.

8.8.2 Pruning orphan files

The opposite situation is when you have a file on disc that is not linked to an entry in the database anymore. This situation could be even more perilous than the previous one and may occur more frequently than you realize. The consequences of this mismatch can be severe, imagine that during the *testing / development phase* of your *Processor* you generate an output figure saved on disc. You then realize that the plot is wrong and you fix the bug and update the DB, but for some reasons you have forgotten to delete the figure file from the disc. Afterwards, while looking for the processor output, you find this file and believe it is a valid result and you use it for your publication. In order to prevent this to happen, you just have to follow some simple rules, and then the reliable machinery of MAFw will do the rest.

The key point is to use a specific trigger in every table that has a file name field. This trigger has to react before any delete query on such a table and inserting all FileNameFields or FileNameListFields in the OrphanFile table. You will see an example of such a trigger in the next paragraphs. This standard tables will be queried by the next processor being executed and during the start super method, all files in the Orphan table will be removed from the disc.

Let us try to understand this better with a step-by-step example. For simplicity, we have removed the import statements from the code snippet, but it should not be too difficult to understand the code anyway.

We begin with the declaration of our input model:

Listing 8.10: File model definition with trigger

```
class File(MAFwBaseModel):
    file_id = AutoField(primary_key=True, help_text='primary key')
    file_name = FileNameField(checksum_field='check_sum', help_text='the file name')
    check_sum = FileChecksumField(help_text='checksum')

    @classmethod
    def triggers(cls) -> list[Trigger]:
        file_delete_file = Trigger(
            'file_delete_file',
            (TriggerWhen.Before, TriggerAction.Delete),
            source_table=cls,
            safe=True,
            for_each_row=True,
        )
        file_delete_file.add_when('1 == (SELECT status FROM trigger_status WHERE␣
↪trigger_type = "DELETE_FILES")')
        file_delete_file.add_sql(OrphanFile.insert(filenames=SQL('OLD.file_name'),␣
↪checksum=SQL('OLD.file_name')))
        return [file_delete_file]

    class Meta:
        depends_on = [OrphanFile]
```

Here you see the trigger definition: it is a before delete type and when triggered it is adding the filename field to the OrphanFile table. It is important to notice that this trigger has a when condition and will only be executed when the trigger type DELETE_FILES is enabled. This is necessary for the pruning mechanism to work, just copy this line in your trigger definition.

And now let us define some fake processors. First we import some files into our model, then we remove some rows from the file table and finally other two processors, doing nothing but useful to demonstrate the effect of the orphan removal.

Listing 8.11: Some example processors

```
@database_required
class FileImporter(Processor):
    input_folder = ActiveParameter('input_folder', default=Path.cwd(), help_doc='From␣
↪where to import')

    def __init__(self, *args, **kwargs):
        super().__init__(*args, looper=LoopType.SingleLoop, **kwargs)
        self.n_files: int = -1

    def start(self):
        super().start()
        self.database.create_tables([File])
        File.delete().execute()
```

(continues on next page)

(continued from previous page)

```
    def process(self):
        data = [(f, f) for f in self.input_folder.glob('**/*dat')]
        File.insert_many(data, fields=['file_name', 'check_sum']).execute()
        self.n_files = len(data)

    def finish(self):
        super().finish()
        if File.select().count() != self.n_files:
            self.processor_exit_status = ProcessorExitStatus.Failed

@database_required
class RowRemover(Processor):
    n_rows = ActiveParameter('n_rows', default=0, help_doc='How many rows to be␣
↪removed')

    def __init__(self, *args, **kwargs):
        super().__init__(*args, looper=LoopType.SingleLoop, **kwargs)
        self.n_initial = 0

    def start(self):
        super().start()
        self.database.create_tables([File])

    def process(self):
        self.n_initial = File.select().count()
        query = File.select().order_by(fn.Random()).limit(self.n_rows).execute()
        ids = [q.file_id for q in query]
        File.delete().where(File.file_id.in_(ids)).execute()

    def finish(self):
        super().finish()
        if File.select().count() != self.n_initial - self.n_rows or OrphanFile.
↪select().count() != self.n_rows:
            self.processor_exit_status = ProcessorExitStatus.Failed

@orphan_protector
@database_required
class OrphanProtector(Processor):
    def __init__(self, *args, **kwargs):
        super().__init__(looper=LoopType.SingleLoop, *args, **kwargs)
        self.n_orphan = 0

    def start(self):
        self.n_orphan = OrphanFile.select().count()
        super().start()

    def finish(self):
        super().finish()
        if OrphanFile.select().count() != self.n_orphan:
            self.processor_exit_status = ProcessorExitStatus.Failed

@single_loop
class LazyProcessor(Processor):
    def finish(self):
        super().finish()
```

(continues on next page)

(continued from previous page)

```
        if OrphanFile.select().count() != 0:
            self.processor_exit_status = ProcessorExitStatus.Failed
```

The **FileImporter**[4] is very simple, it reads all dat files in a directory and loads them in the File model along with their checksum. Just to be sure we empty the File model in the start and in the finish we check that the number of rows in File is the same as the number of files in the folder.

The **RowRemover** is getting an integer number of rows to be removed. Even though the File model is already created, it is a good practice to repeat the statement again in the start method. Then we select some random rows from File and we delete them. At this point, we have created some orphan files on disc without related rows in the DB. Finally (in the finish method), we verify that the number of remaining rows in the database aligns with our expectations. Additionally, we ensure that the trigger functioned correctly, resulting in the appropriate rows being added to the OrphanFile model.

The **OrphanProtector** does even less than the others. But if you look carefully, you will see that along with the *database_required()* there is also the *orphan_protector()* decorator. This will prevent the processor to perform the check on the OrphanFile model and deleting the unrelated files. In the start method, we record the number of orphan files in the OrphanFile model and we confirm that they are still there in the finish. Since the actual removal of the orphan files happens in the processor start method, we need to count the number of orphans before calling the super start.

The **LazyProcessor** is responsible to check that there are no rows left in the OrphanFile, meaning that the removal was successful.

And now let us put everything together and run it.

Listing 8.12: ProcessorList execution

```
db_conf = default_conf['sqlite']
db_conf['URL'] = 'sqlite:///:memory:'
plist = ProcessorList(name='Orphan test', description='dealing with orphan files',␣
↪database_conf=db_conf)
importer = FileImporter(input_folder=tmp_path)
remover = RowRemover(n_rows=n_delete)
protector = OrphanProtector()
lazy = LazyProcessor()
plist.extend([importer, remover, protector, lazy])
plist.execute()

assert importer.processor_exit_status == ProcessorExitStatus.Successful
assert remover.processor_exit_status == ProcessorExitStatus.Successful
assert protector.processor_exit_status == ProcessorExitStatus.Successful
assert lazy.processor_exit_status == ProcessorExitStatus.Successful
```

In practice, the only thing you have to take care of is to add a dedicated trigger to each of your tables having at least a file field and then the rest will be automatically performed by MAFw.

Warning

You should be very careful if your processor is removing rows from the target table (where you should be storing the processor's results). This might be the case of a processor that wants to reset the status of your analysis to a previous step, for example. In this case, as soon as *ProcessorA* removes the rows from the model, the trigger will inserts all FileNameFields in the OrphanFile model in order to be deleted. This is a lazy operation and will be performed by the following processor to be executed either in the same pipeline or in the next. When *ProcessorA* will have finished its work, the target table will be repopulated and the same will happen to the

[4] A much better implementation of an Importer could be achieved using a subclass of the *Importer*. See, for example, the *ImporterExample* class and its *documentation*.

folders on the disc. Now the next processor will empty the orphan file model and possibly remove the freshly generated files.

You have two solutions for this problem: either you block the execution of the trigger when deleting the rows (you can use the *TriggerDisabler* for this purpose), in this way the rows in the model will be removed, but not the files from disc with the risks we have already mentioned. The second possibility is to force the processor to immediately take care of the orphan file pruning. This is the suggested procedure and you only need to include a call to the *_remove_orphan_files()* soon after the delete query.

Automatic creation of *file removal* trigger

In the definition of *File model* we have manually created a trigger that is fired every time a row is removed from the table. The action of this trigger is to insert in the *OrphanFile* model the content of the *FileNameField* that is actually the specifically designed Field type to store a path to a file.

This trigger creation part can be totally automatized, if you want to insert in the *OrphanFile* the content of all *FileNameField* and *FileNameListField* fields. The automatic creation is another advantage of using the *MAFwBaseModel*. This functionality can be activated very simply by setting a flag in the metadata class.

Listing 8.13: File model definition with automatic trigger

```
class File(MAFwBaseModel):
    file_id = AutoField(primary_key=True, help_text='primary key')
    file_name = FileNameField(checksum_field='check_sum', help_text='the file name')
    check_sum = FileChecksumField(help_text='checksum')

    class Meta:
        depends_on = [OrphanFile]
        file_trigger_auto_create = True
```

This model class is absolutely identical to the *previous one*; MAFw base model will take care of generating the necessary trigger.

It is worth remembering that *file_trigger_auto_create* is inherited by subclasses, so if you want all the models in your project to have this feature, you can simply set it to true in your base model.

Warning

We have already mentioned the main disadvantage of working with triggers, that is to say the difficult debugging. Triggers are pieces of code that are executed in full autonomy by the database, so when you are trying to understand why your analysis pipeline is not working as expected a part from looking at your python code, you should also keep in mind that triggers might be the source of the problem.

This effect is amplified, if you turn on the automatic trigger creation, because you don't even see the trigger definition in your python code and you may have forgotten about the auto trigger flag. As always with great power comes even greater responsibility!

8.8.3 Keeping the entries updated

One aspect is that the file exists; another is that the file content remains unchanged. You may have replaced an input file with a newer one and the database will not know it. If your processors are only executed on items for which there is still no output generated, then this replaced file may go unnoticed causing issues to your analysis.

For this reason, we are strongly recommending to always add a checksum field for each file field in your table. Calculating a checksum is just a matter of a split second on modern CPU while the time for the debugging your analysis code is for sure longer.

The function *verify_checksum()* takes a Model as argument and will verify that all checksums are still valid. In other words, for each FileNameField (or FileNameListField) with a link to a checksum field in the table, the

function will compare the actual digest with the stored one. If it is different, then the DB row will be removed.

Also this function is not automatically called by any processor super methods. It is ultimately the user's responsibility to decide whether to proceed, bearing in mind that working with long tables and large files may result in delays in processor execution.

The implementation is very similar to the previous one, just change the function name. Keep in mind that the *verify_checksum()* will implicitly check for the existence of files and warn you if some items are missing, so you can avoid the *remove_widow_db_rows()*, if you perform the checksum verification.

```
class MyProcessor(Processor):

    def start():
        super().start()
        verify_checksum(InputData)
```

8.9 Multi-primary key columns

Special attention is required when you need to have a primary key that is spanning over two or more columns of your model. So far we have seen how we can identify one column in the model as the primary key and now we will see what to do if you want to use more than one column as primary key and, even more important, how you can use this composite primary key as a foreign key in another model.

To describe this topic, we will make use of an example that you can also find in the examples modules of MAFw named *multi_primary*.

Let us start with the model definition.

```
class Sample(MAFwBaseModel, do_not_register=True):
    sample_id = AutoField(primary_key=True, help_text='The sample id primary key')
    sample_name = TextField(help_text='The sample name')

class Resolution(MAFwBaseModel, do_not_register=True):
    resolution_id = AutoField(primary_key=True, help_text='The resolution id primary key')
    resolution_value = FloatField(help_text='The resolution in µm')

class Image(MAFwBaseModel, do_not_register=True):
    image_id = AutoField(primary_key=True, help_text='The image id primary key')
    filename = FileNameField(help_text='The filename of the image', checksum_field='checksum')
    checksum = FileChecksumField()
    sample = ForeignKeyField(Sample, on_delete='CASCADE', backref='+', lazy_load=False, column_name='sample_id')
    resolution = ForeignKeyField(
        Resolution, on_delete='CASCADE', backref='+', lazy_load=False, column_name='resolution_id'
    )

class ProcessedImage(MAFwBaseModel, do_not_register=True):
    image = ForeignKeyField(
        Image,
        primary_key=True,
        backref='+',
        help_text='The image id, foreign key and primary',
        on_delete='CASCADE',
        lazy_load=False,
        column_name='image_id',
    )
```

(continues on next page)

(continued from previous page)

```
    value = FloatField(default=0)

class CalibrationMethod(MAFwBaseModel, do_not_register=True):
    method_id = AutoField(primary_key=True, help_text='The primary key for the calculation method')
    multiplier = FloatField(default=1.0, help_text='The multiplication factor of this method')

class CalibratedImage(MAFwBaseModel, do_not_register=True):
    image = ForeignKeyField(
        ProcessedImage,
        on_delete='CASCADE',
        help_text='The reference to the processed image',
        backref='+',
        lazy_load=False,
        column_name='image_id',
    )
    method = ForeignKeyField(
        CalibrationMethod,
        on_delete='CASCADE',
        help_text='The reference to the calibration method',
        backref='+',
        lazy_load=False,
        column_name='method_id',
    )
    calibrated_value = FloatField(default=0.0, help_text='The calibrated value')

    @property
    def primary_key(self) -> Iterable:
        return self.image_id, self.method_id

    class Meta:
        primary_key = CompositeKey('image', 'method')

class ColoredImage(MAFwBaseModel, do_not_register=True):
    image_id = IntegerField(help_text='The reference to the processed image. Combined FK with method_id')
    method_id = IntegerField(help_text='The reference to the calibration method. Combined FK with method_id')
    red = FloatField(default=0, help_text='Fake red. Only for testing')
    green = FloatField(default=0, help_text='Fake green. Only for testing')
    blue = FloatField(default=0, help_text='Fake blue. Only for testing')

    @property
    def primary_key(self) -> Iterable:
        return self.image_id, self.method_id

    class Meta:
        constraints = [
            SQL(
                'FOREIGN KEY (image_id, method_id) REFERENCES '
                'calibrated_image(image_id, method_id) ON DELETE CASCADE'
            )
        ]
        primary_key = CompositeKey('image_id', 'method_id')

```

As always, one single picture can convey more than a thousand lines of code. Here below the ERDs of Image and of CalibratedImage.

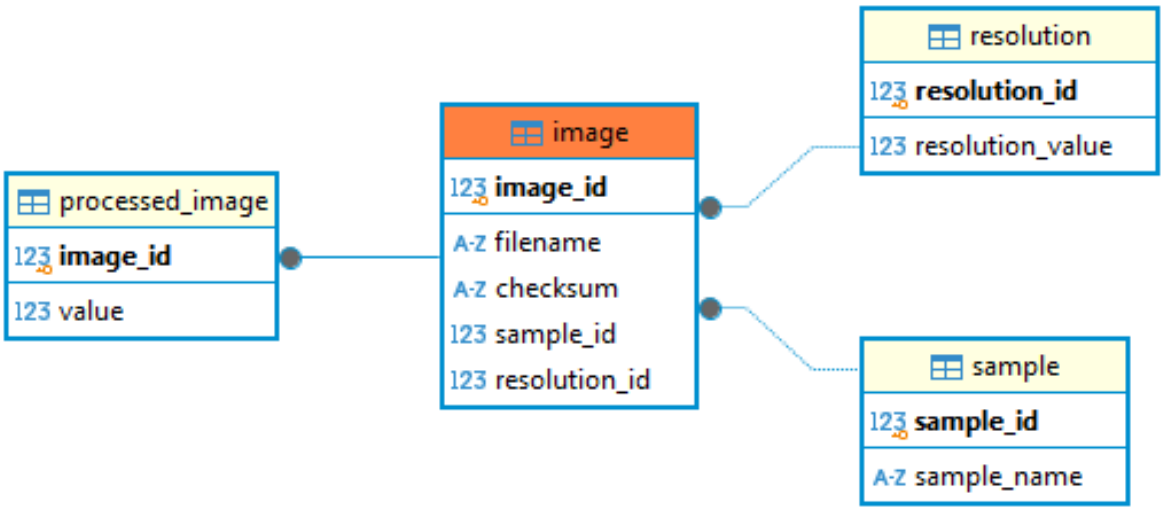


Fig. 8.1: The ERD of the Image Model

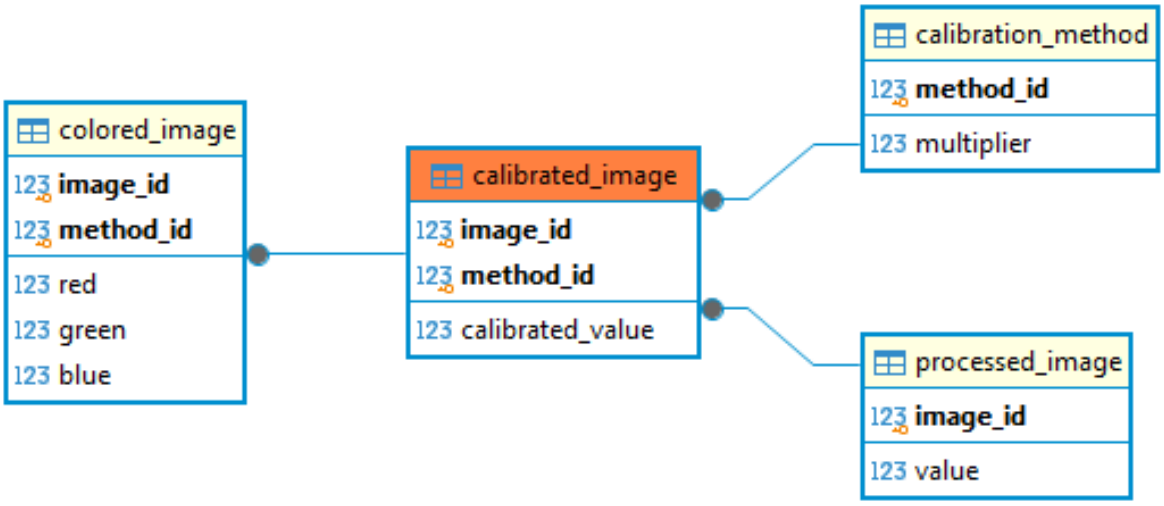


Fig. 8.2: The ERD of the CalibratedImage Model

In the diagrams, the fields with bold font represent primary keys, also highlighted by the separation line, while the arrow are the standard relation.

As in the examples above, we have images of different samples acquired with different resolutions entering the Image model. We use those lines to make some calculations and we obtain the rows in the ProcessedImage model. These two tables are in 1 to 1 relation and this relation is enforced with a delete cascade, meaning that if we delete an element in the Image model, the corresponding one in the ProcessedImage will also be deleted.

The CalibrationMethod model contains different sets of calibration constants to bring each row from the ProcessedImage model to the CalibratedImage one. It is natural to assume that the `image_id` and the `method_id` are the best candidates to be a combined primary key. To achieve this, in the CalibratedImage model, we need to add (line 57-58) an overload of the Meta class, where we specify our `CompositeKey`. Pay attention to an important detail: the CompositeKey constructor takes the name of the fields and not the name of the columns, that in the case of foreign keys differ of '_id'. Optionally we can also define a primary_key property (line 53-55) to quickly retrieve the values of our keys.

From the application point of view, we want all the processed images to be calibrated with all possible calibration methods, that means we need to make a cross join as described below:

```
with database.atomic() as txn:
    cross_join = (
        ProcessedImage.select(ProcessedImage, CalibrationMethod).
↪join(CalibrationMethod, JOIN.CROSS).execute()
    )
    for row in cross_join:
        calibrated_image = CalibratedImage()
        calibrated_image.image_id = row.image_id
        calibrated_image.method_id = row.method_id
        calibrated_image.calibrated_value = row.value * row.multiplier
        calibrated_image.save(force_insert=True)
```

Up to this point we have seen what we have to do to specify a composite primary key, we cannot use the AutoField or the primary_key parameter, but we need to go through the Meta class in the way shown in the example.

The next step is to have another table (ColoredImage in our imaginary case) that is in relation with CalibratedImage. We would need to have again a composite primary key that is also a composite foreign key. Peewee does not support composite foreign keys, but we can use the workaround shown at lines 72-77. Along with the CompositeKey definition, we need to add a Constraint as well using the SQL function to convert a string into a valid SQL statement. This time, since we are using low level SQL directives, we have to use the column names (additional '_id') instead of the field name.

And in a similar way we can insert items in the ColoredImage model.

```
with database.atomic() as txn:
    full_list = CalibratedImage.select().execute()

    for row in full_list:
        colored_image = ColoredImage(image_id=row.image_id, method_id=row.method_id)
        colored_image.red = row.calibrated_value / 3
        colored_image.green = row.calibrated_value / 3
        colored_image.blue = row.calibrated_value / 3
        colored_image.save(force_insert=True)
```

Now, with all the tables linked to each other, try to delete one from a table, and guess what will happen to all other tables.

This tutorial might be a bit more complex than the examples we have seen so far, but we believed you have appreciated the power of such a relational tool.

8.10 Importing an existing DB

The last section of this long chapter on database will show you how to deal with an existing DB. It is possible that before you have adopted MAFw for your analysis tasks, you were already employing a relational database to store your dataset and results. So far we have seen how to create tables in a database starting from an object oriented description (a model) in a python library. But what do we have to do if the database already exists? Can we create the classes starting from a database? This process goes under the name of **reflection** and it is the subject of this section.

The reflection of tables in python classes cannot be performed automatically at 100% by definition. A typical case is the use of application specific fields. Consider, for example, the FileNameField that we have discussed earlier. This field corresponds to a Path object when you look at it from the application point of view, but the actual path is saved as a text field in the concrete database implementation. If you now read the metadata of this table from the database point of view, you will see that the field will contain a text variable and thus the reflected class will not have any FileNameField.

Let us try to understand the process looking at the picture below. If we create the model in python, then we can

assign special field descriptors to the table columns, but their concrete implementation in the database must be done using types that are available in the database itself. So when we perform the reverse process, we get only a good approximation of the initial definition.

```python
class Image(MAFwBaseModel):  14 usages (2 dynamic)  BULGHERONI Antonio *
    image_id = AutoField(primary_key=True, help_text='The image id primary key')
    filename = FileNameField(help_text='The filename of the image', checksum_field='checksum')
    checksum = FileChecksumField()
    sample = ForeignKeyField(Sample, on_delete='CASCADE', backref='+', lazy_load=False, column_name='sample_id')
    resolution_id = ForeignKeyField(
        Resolution, on_delete='CASCADE', backref='+', lazy_load=False, column_name='resolution_id'
    )
```

Fig. 8.3: This is the model implementation as you would code it making use of the specific field definitions.

Column Name	#	Data Type
123 image_id	1	INTEGER
A-Z filename	2	TEXT
A-Z checksum	3	TEXT
123 sample_id	4	INTEGER
123 resolution_id	5	INTEGER

Fig. 8.4: During the actual implementation of the model as a database table, python column definitions will be translated into database types.

```python
class Image(MAFwBaseModel):  3 usages (2 dynamic)
    image_id = AutoField()
    filename = TextField()
    checksum = TextField()
    sample = ForeignKeyField(column_name='sample_id', field='sample_id', model=Sample)
    resolution = ForeignKeyField(column_name='resolution_id', field='resolution_id', model=Resolution)

    class Meta:
        table_name = 'image'
```

Fig. 8.5: The reflection process will translate the database implementation in a generic model implementation, not necessarily including all the specific field definition.

Nevertheless the process is rather efficient and can generate an excellent starting point that we can use to customize the model classes to make them more useful in our application.

From a practical point of view, you just have to open a console and type the command `mafw db wizard --help` to get some help on the tool and also read its documentation. You need to provide the name of the database and how to connect to it, in the case of Sqlite DB, it is enough to provide the filename, and you have to specify the name of the output python file that will contain all the model classes. This module is ready to go, you could theoretically import it into your project and use it, but it is strongly recommended to accurately check that everything is really the way you want it to be.

The reflection process is absolutely safe for your existing database, so it is worth to give it a try!

8.11 Execute SQL scripts

If you are not new to databases, you might have some SQL script files hanging around. Something link initialisation or optimisation procedure. If this is the case, you can include those in your analysis pipelines.

The *SQLScriptRunner* processor provides a convenient way to execute SQL scripts against your database. This is particularly useful when you need to perform database operations that are not easily expressed through the ORM or when you want to leverage database-specific features.

To use the SQLScriptRunner, you need to configure it in your steering file by specifying a list of SQL files to be processed. Here's an example configuration:

```
[SQLScriptRunner]
sql_files = ["./scripts/init_schema.sql", "./scripts/populate_data.sql"]
```

The processor will:

1. Validate that all specified SQL files exist and are regular files

2. Read each SQL file content
3. Remove multi-line block comments (/* … */) from the SQL content
4. Split the content into individual SQL statements
5. Execute all statements within a single atomic transaction

This ensures that either all statements are executed successfully, or none are applied if an error occurs, maintaining database integrity.

Each SQL file should contain one or more valid SQL statements separated by semicolons. The processor automatically handles the transaction boundaries, so you don't need to include explicit BEGIN/COMMIT statements in your SQL files.

Warning

Since all statements are executed within a single transaction, very large SQL files or operations that take a long time to complete might impact database performance. Consider breaking large scripts into smaller chunks if needed.

8.12 What's next

Congratulations! You reached the end of the most difficult chapter in this tutorial. It is difficult because as a scientist you might not be used to deal with databases everyday, but their power is incredible, isn't it?

The next chapter is about efficiency! You will learn how to process only the data you want and that need to be analysed. Get ready to understand database filters!

CHAPTER 9

FILTERS: DO ONLY WHAT IS NEEDED TO BE DONE!

As mentioned already several times, the main role of the database is to support the processor execution in providing the input items and in storing the output products. Not everything can be efficiently stored in a database, for example large chunk of binary data are better saved to the disc, in this case, the database will know the path where these data are stored.

One advantage of the database is that you can apply selection rules and you do not have to process the whole dataset if you do not need it. To help you in this, MAFw is offering you a ready-to-use solution, the *db_filter*.

This module is providing a filtering system that allows users to precisely control which data is processed by each processor. This system offers several key benefits:

- **Precision Control**: Users can define exactly which records should be included or excluded from processing
- **Performance Optimization**: By filtering early in the pipeline, unnecessary computations are avoided
- **Flexibility**: Supports various filtering operations including simple conditions, logical combinations, and conditional logic
- **Configuration-Driven**: Filters can be easily configured through the steering files without code changes
- **Scalability**: Complex filtering scenarios can be expressed through logical expressions and nested conditions

The filtering system is built around three main components plus an extra one:

- *ProcessorFilter* - For managing multiple model filters in a processor
- *ModelFilter* - For defining filters at the model level
- *ConditionNode* - For defining filters at the field level
- *ConditionalFilterCondition* - For conditional filtering logic (this is the extra)

and they implement the following hierarchical structure:

9.1 How to use a filter

The filter native playground is in the implementation of the *get_items()* method and of course it can only work if the list of items is retrieved from the database.

Let us assume that our *Processor*, named AdvProcessor, is using three models to obtain the item lists. Everything is nicely described in the ERD below. The three models are interconnected via foreign key relations. There is a fourth model, that is where the output data will be saved.

The real core of our database is the image table, where our data are first introduced, the other two on the right, are kind of helper tables storing references to the samples and to the resolution of our images. The processed_image table is where the output of our AdvProcessor will be stored.

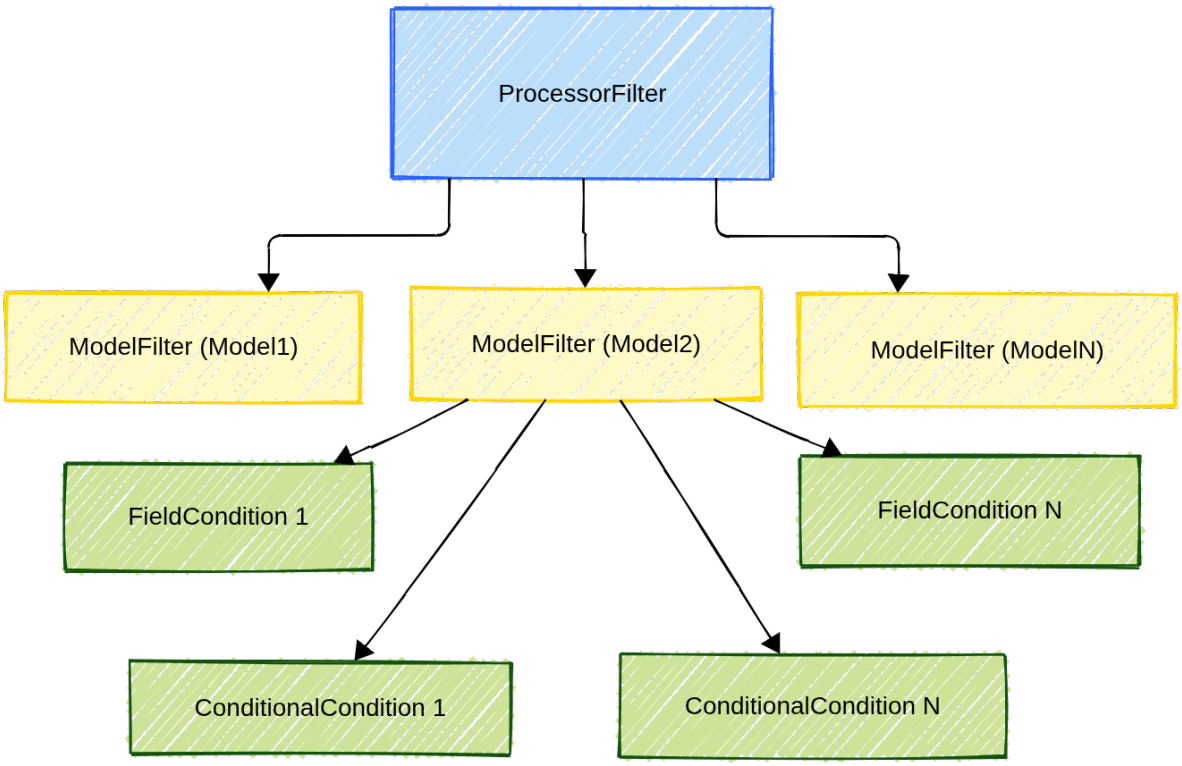


Fig. 9.1: The description of the filtering hierarchical structure.

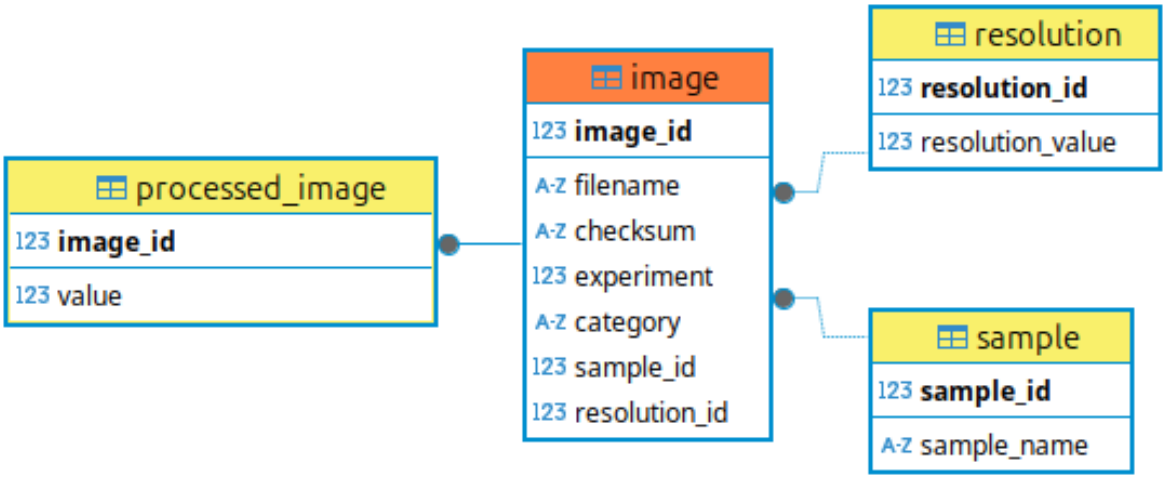


Fig. 9.2: The ERD of the AdvProcessor.

To realize this database with our ORM we need to code the corresponding model classes as follows:

```
class Sample(MAFwBaseModel):
    sample_id = AutoField(primary_key=True, help_text='The sample id primary key')
    sample_name = TextField(help_text='The sample name')

class Resolution(MAFwBaseModel):
    resolution_id = AutoField(primary_key=True, help_text='The resolution id primary␣
↪key')
    resolution_value = FloatField(help_text='The resolution in µm')

class Image(MAFwBaseModel):
    image_id = AutoField(primary_key=True, help_text='The image id primary key')
    filename = FileNameField(help_text='The filename of the image', checksum_field=
↪'checksum')
    checksum = FileChecksumField(help_text='The checksum of the input file')
    experiment = IntegerField(default=1,
                              help_text='A flag for selection of the experiment.␣
↪Flags are bitwise combinable')
    category = TextField(
        help_text='A text string to describe the image category. Accepted values are:
↪"STANDARD", "SUT", "REFERENCE"')
    sample = ForeignKeyField(
        Sample, Sample.sample_id, on_delete='CASCADE', backref='sample', column_name=
↪'sample_id'
    )
    resolution = ForeignKeyField(
        Resolution, Resolution.resolution_id, on_delete='CASCADE', backref='resolution
↪', column_name='resolution_id'
    )

class ProcessedImage(MAFwBaseModel):
    image = ForeignKeyField(
        Image,
        Image.image_id,
        primary_key=True,
        column_name='image_id',
        backref='raw',
        help_text='The image id, foreign key and primary',
        on_delete='CASCADE',
    )
    value = FloatField(default=0)
```

By now, you should be an expert in ORM and everything there should be absolutely clear, otherwise, take your chance now to go back to the previous sections or to the peewee documentation to find an explanation. Note how the Image class is making use of our *FileNameField* and *FileChecksumField*. We added also a bit of help text to each field, in order to make even more evident what they are.

Particularly interesting is the experiment field in the Image model. This is a binary flag and can be very useful to assign one file (an image in this case) to one or more experiments. For example, imagine you have three different experiments in your data analysis; you assign to the first experiment the label 1 (binary: 0b1), to the second the label 2 (0b10) and to the third the label 4 (0b100). Now, if you want an image to be used only for experiment 1, you set the experiment column to 1; similarly if you want an image to be part of experiment 1 and 3, then you set its experiment column to 1 + 4 = 5 (b101). In fact, if you bit-wise AND, this image with the label of the experiments (5 BIT_AND 1 = 5 BIT_AND 4 = True) you will get a True value.

For each foreign key field, we have specified a backref field, so that you could get access to the related models. Pay attention also at the highlighted lines, where we define foreign key fields to other tables. Peewee follows Django style references, so actually the field object is named with the noun of the object you are referring to. This will

allow the following:

```
image.sample # Resolve the related object returning a Sample instance, it costs an␣
↪additional query
image.sample_id # Return the corresponding Sample's id number
```

The primary source of input is the Image; however, you may wish to process only images that meet specific criteria, such as belonging to a particular sample or being captured at a certain resolution. Unfortunately, this information is not explicitly included in the Image model. Only the resolution_id and the sample_id are included in the image table: those primary keys are excellent for a computer, but for a human being it is better to use sample names and resolution values. The solution is to use a join query in order to have all fields available and then we will be able to apply the configurable filters from the TOML steering file to limit the selection to what we want.

Since we have three input models for our processor we can have up to three different Model level filters defined in the steering files. The processor is storing all the filters configured in the steering files in a container class *filter register* that will be the main actor of our *get_items()* implementation that follows:

```
def get_items(self):

    # first of all, let us be sure that the tables exist
    # the order is irrelevant, the ORM will find the best creation strategy.
    # if the table already exists, nothing will happen.
    self.database.create_tables([Sample, Resolution, Image, ProcessedImage])

    # if you want to remove widow rows from the output table or verify the checksum␣
↪do it now!
    remove_widow_rows([Image, ProcessedImage])

    # did we select new_only in the global filter?
    # if yes, prepare an additional condition in which we specify that the
    # Image.image_id should not be among the image_id of the ProcessedImage.
    # if no, then just accept everything.
    if self.filter_register.new_only:
        # let us get a list of all already processed items.
        # since the item in the output table are stored using the same primary key,
        # this will simplify the discrimination between new and already processed␣
↪items.
        existing_entries = ProcessedImage.select(ProcessedImage.image_id).execute()

        # build the condition
        existing = ~Image.image_id.in_([i.image_id for i in existing_entries])
    else:
        existing = True

    # finally let us make the query.
    query = (Image.select(Image, Sample, Resolution)
                  .join(Sample, on=(Image.sample_id == Sample.sample_id), attr='s')
                  .switch(Image)
                  .join(Resolution, on=(Image.resolution_id == Resolution.resolution_
↪id), attr='r')
                  .where(self.filter_register.filter_all())
                  .where(existing)
             )
    return query
```

The processor filter register comes up in two points (highlighted in the code), when can retrieve a boolean flag representing whether we want to process input data for which an output record already exists or not.

The second usage is in the query, where we generate a peewee combined expression representing the whole combination of all user defined (and configured) filters. Let's move on and see how we configure a filter!

9.2 How to configure a filter

In a steering file, there is a table for each processor where you can set the configurable active parameters. To this table, you can add a sub-table named *__filter__*, containing other tables, one for each input Model. The reason for the underscores is to avoid to pollute the processor parameter namespace. This is how it will look like:

```
[AdvProcessor]
param1 = 15

[AdvProcessor.__filter__.Resolution]
resolution_value = 25

[AdvProcessor.__filter__.Sample]
sample_name = 'sample_00*'
```

In the example above, AdvProcessor has two *.__filter__.* tables, one for Resolution and one for Sample. When the steering file will be parsed, the processor constructor will automatically generate two filters: for Resolution it will put a condition that the resolution field must be 25 and for Sample, the sample_name column should be 'sample_1'. You could add a third table for the Image model, if you wish.

In this way, the processor initialization machinery will create the necessary filters and add them to the processor filter register from where we can retrieve it.

The *new_only* parameter of the register can also be configured in the steering file. By default, you can find in the steering file in the general section a parameter *new_only = true*. This flag will be applied to all processors in the pipeline, but you can change it for each processor in this way:

```
[AdvProcessor]
__new_only__ = false
param1 = 15

# The rest as before...
```

In this way, for the AdvProcessor the new only flag will be False[1], while the rest is left to the default value.

If we use this configuration the AdvProcessor will obtain an input item list with only the input images having a resolution_value of 25 **AND** a sample_name matching the glob style string sample_00*.

It is important to underline that the filters on the models are joined (by default) using an AND. In this simple example we had one field for each model being the subject of the filter, but you could have as many as you like. Inside a model, the default joining logic is again **AND**.

The configuration of these filters is done using the so called *single value* approach, even if for the sample_name we specified an * in the name. The reason is that we have assigned with the = symbol one single value to the model field.

The filter will interpret the meaning of = differently depending on the type of the right hand side value, following this table:

Filter field type	Logical operation	Example
Numeric, boolean	==	Field == 3.14
String	GLOB	Field GLOB '*ree'
List	IN	Field IN [1, 2, 3]

[1] Remember that in TOML, the two booleans are **NOT** capitalized as in Python. Moreover, you can specify new_only also in Processor/Model but it will not be taken into account unless that model has a column named new_only.

9.3 Explicit operation configuration

In the previous example we have seen how to select one specific field to be exactly equal to a given value, but maybe our goal is to select an interval, or performing a bitwise logical operation. The filter system also supports explicit operation specification, allowing you to define the exact logical operation to be applied. Here is an example:

```
[AdvProcessor]
param1 = 15

[AdvProcessor.__filter__.Resolution]
resolution_value = {op = ">=", value = "25"}

[AdvProcessor.__filter__.Image]
experiment = { op = "BIT_AND", value = 5 }
```

In this configuration example, you can see to practical usage of the so-called explicit operation configuration. It is called explicit, because this time you clear specify which operation you want to apply. So, in this case we will be getting all images with a resolution value greater or equal to 25 and an experimental flag returning true when bitwise AND'ed to 5. If you remember, in our introduction, this is equivalent to select experiment labelled with 1 and with 4.

The supported operations include[2]:

Operation	Description	Example
==	Equal	Field == 42
!=	Not equal	Field != 42
<	Less than	Field < 100
<=	Less than/equal	Field <= 100
>	Greater than	Field > 0
>	Greater than/equal	Field >= 10
GLOB	Pattern matching	Field GLOB 'test*'
LIKE	SQL LIKE	Field LIKE 'test%'
REGEXP	Regular expression	Field REGEXP '^[A-Z]'
IN	Value in list	Field IN [1, 2, 3]
NOT_IN	Value not in list	Field NOT_IN [1, 2]
BETWEEN	Value between	Field BETWEEN [1, 10]
BIT_AND	Bitwise AND	Field & 5 != 0
BIT_OR	Bitwise OR	Field | 7 != 0
IS_NULL	Field is NULL	Field IS NULL
IS_NOT_NULL	Field is not NULL	Field IS NOT NULL

Note

The default sqlite library provides only an abstract definition of the regular expression matching. In simple words, it means that the user needs to implement the user function *regexp()*, or using any sqlite extensions that implements it.

In summary, if you are using the vanilla sqlite, you **cannot use the REGEXP operator** in your filter and you need to reformulate your filtering condition using a combination of other string matching tools.

[2] The list of all overloaded operator and special method is available in the peewee doc.

9.4 Logical Composition

In the examples seen so far, the field conditions inside of a *ModelFilter* were logically AND'ed to generate the condition for that specific model. If more model filters are present, then the overall processor filter was generated logically AND'ing all model filters. Always keep in mind this hierarchical structure: Field Condition < Model Condition < Processor Condition described in Fig. 9.1.

You may want to have a different logical composition, defining how field conditions inside a model or model filters inside a processor are combined. To do so use the *__logic__* keyword to define the logical composition. See the example below.

Listing 9.1: Logical composition filtering

```
[AdvProcessor]
param1 = 15
__logic__ = "(Resolution OR Sample) AND Image"

[AdvProcessor.__filter__.Resolution]
resolution_value = {op = ">=", value = "25"}

[AdvProcessor.__filter__.Image]
__logic__ = "category OR image_id"
experiment = { op = "BIT_AND", value = 5 }
category = "beautiful_images"

[AdvProcessor.__filter__.Image.image_id]
__logic__ = "good_images AND NOT bad_images"
good_images = {op = "BETWEEN", value = [100, 145]}
bad_images = {op = "IN", value=[105, 109]}

[AdvProcessor.__filter__.Sample]
sample_name = "sample_00*"
sample_id = [1,2]
```

This is a very advanced configuration and I hope you will appreciate how easy is to obtain.

First of all there is a logic sentence in the AdvProcess table. It says that we want to combine the expressions for the Resolution and the Sample models using OR. The result of this should be combined with AND with the expression for the Image model.

The resolution filter is an explicit condition as seen in the previous paragraph. There is only one condition for this model, so nothing to combine. For the Sample model, we have two field conditions and no __logic__ keyword, so the two conditions will be combined with the default AND.

The filter on Image is a bit more complicated. First of all, it has a logic statement that will be used to combine the field condition. The statement is mentioning *category* and *image_id*, but nothing about *experiment*, as a consequence the condition on the experiment will be simply ignored. The category condition is a simple value one while the one over *image_id* is a nested. In fact for *image_id*, we have an additional table describing what we want. This is a combination governed by a logic statement where we have defined two sub-conditions, a range of good images to be combined with some bad images.

This whole configuration is equivalent to this SQL statement:

```
SELECT
    *
FROM
    image
    join resolution using(resolution_id)
    join sample using(sample_id)
WHERE
```

(continues on next page)

(continued from previous page)

```
    ( resolution.resolution_value >= 25 OR
      (sample.sample_name GLOB "sample_00*"
        AND sample.sample_id IN (1, 1) ) AND
    ( image.image_id BETWEEN 100 and 145 AND NOT image.image_id IN (105, 109))
```

Summary

There are three different filtering levels: Field, Model, Processor.

For each level, you can define as many conditions as you wish. Those conditions are by default combined using **AND**, but more advanced logical composition can be achieved using the *__logic__* keyword via a simplify grammar.

The logical grammar uses only three logical operators: **AND**, **OR** and **NOT**. The variable names are the model names at the processor level, the field name at the model level and the condition name at the field level.

Be aware that the grammar is **case sensitive**: the operators must always be written in capitol letters, while the variable names should respect the original case. Use parenthesis to specify the order. When a logic statement is provided, variables that are not mentioned in the statement will not be included in the filter generation.

Each condition lives in its own scope and cannot be directly linked to conditions in other scope. So for example, field conditions of defined in Resolution cannot be included in the logic statement of the Image model.

9.5 Conditional filters

In the introduction, we mention the three pillars on which the filtering system is based plus one extra. This extra one is the conditional filter! They allow expressing logic where one field's criteria depend on another field's value.

They live inside the model scope since they relate different fields of the same table. For each model, you can have as many as you like and you can also include them in the model level logic statement.

Listing 9.2: Conditional filtering

```
[AdvProcessor]
param1 = 15

[AdvProcessor.__filter__.Image]
__logic__ = "resolution_id AND conditional_1"
resolution_id = 1 # this is the foreign key value

[[AdvProcessor.__filter__.Image.__conditional__]] # this is a list, note the double␣
↪[]
name = 'conditional_1'
condition_field = 'category'
condition_op = 'GLOB'
condition_value = 'beautiful'
then_field = 'raw_image_id'
then_op = 'BETWEEN'
then_value = [100, 140]
else_field = 'raw_image_id' # OPTIONAL
else_op = '>' # OPTIONAL
else_value = 200 # OPTIONAL
```

The conditional filter translates as follow: if the category is matching 'beautiful', then select raw_image_id between 100 and 140, otherwise (if not matching 'beautiful') select raw_image_id greater than 200.

The SQL equivalent is:

```
WHERE resolution_id = 1 AND
     (( category GLOB "beautiful" AND raw_image_id BETWEEN 100 AND 140 )
     OR ( category NOT GLOB "beautiful" AND raw_image_id > 200 ))
```

Summary

Conditional filters allows to select one field based on the value of another fields.

You can add as many as you like, just repeat the same table header (with the double set of []).

9.6 Filter schema metadata, your help to configure filters

We have seen how powerful and versatile filters are, but you have also seen how tricky it can be to configure them. In the future, we are planning to release a GUI based tool for the generation of steering files, this will simplify a lot the editing part and also help in the generation of filter conditions. In order for this to work we need to define a contract between a processor and the models in the database where the processor would like to be able to operate filters. This *contract* is optional, but implement it costs you really just one line of code and will pay back a lot in the future. If you do not sign the contract, your processors will keep on working as before, but the GUI tool will not be able to help you in writing the filter definitions. The static filter metadata is the contract and allows GUI builders to introspect the models involved in a processor without executing the query. The optional *filter_schema()* classmethod returns a *FilterSchema* describing the root table that drives the filtering logic and any supplementary models that can be joined. Because the schema is accessed as a class attribute there is no need to instantiate the processor, and the metadata remains purely declarative.

```
from peewee import CharField

from mafw.db.db_model import MAFwBaseModel
from mafw.decorators import single_loop
from mafw.processor import Processor
from mafw.models.filter_schema import FilterSchema

class Activity(MAFwBaseModel):
    label = CharField()

    class Meta:
        table_name = 'filter_schema_activity'

class Tag(MAFwBaseModel):
    name = CharField()

    class Meta:
        table_name = 'filter_schema_tag'

class ActivityProcessor(Processor):
    _filter_schema = FilterSchema(root_model=Activity, allowed_models=(Tag,))

schema = ActivityProcessor.filter_schema()
assert schema is not None
assert schema.allowed_models[0]._meta.table_name == 'filter_schema_tag'
```

Frontend tooling can use `schema.root_model` and `schema.allowed_models` to explore `_meta.fields`, build drop-downs, and surface join paths—all without touching `get_items()` or firing SQL.

Adding _filter_schema in the class definition of your processor is equivalent to inform the GUI helper tools (still to be released) that the processor may apply filters on *Activity* and on *Tag* as well.

9.6.1 Disabling a filter

Configuring a filter involves a lot of TOML file editing. TOML files are human-readable, but not exactly human-friendly! Sometimes you may want to have filters that can be easily disabled, the most typical example is when you are debugging a new processor. You do not want to run the new analysis over the whole dataset, but a few selected items are more than enough for your test. Once you have set up this filter, you may want to enable and disable it depending on the outcome of your debug.

The easiest way to achieve this is to use the *__enable__* boolean flag in the filter. All levels of filters accept this field, that is true by default. Disabling a ModelFilter is disabling also all children filters (FieldFilter and ConditionalFilter) due to the hierarchical nature of MAFw filtering system.

> **Note**
>
> The *__enable__ = false* in the steering file is equivalent of commenting the whole section, but it is the advantage of leaving the configuration present in the steering file. This is particularly useful when you do not edit the steering file directly as text, but you use the graphical user interface to edit the file.

9.7 How to use the *new_only* filter

9.7.1 Basic usage example

The basic usage of the new only filter is based on the relation between input and output table. Going back to our previous *example*, we have an Image model containing our input data and a ProcessedImage model containing the output of the processor. The two models are in a 1-to-1 relation via a foreign key constrain linking the primary keys of both models. In this case the selection of the *so-called* new_only elements, it is to say the items in the input list for which an output is not yet existing is very easy to obtain. You get the whole list of primary keys in the output table (this is the existing list) and you compare with the list of primary keys in the input table. If the two lists are different, then run the processor only on the difference!

> **Note**
>
> The output primary key list is at most equal to the input primary key. It cannot contain elements that are not in the input table by definition of 1-to-1 relation.

Programmatically you can obtain this in the implementation of the `processor.Processor.get_items()` method as shown in *this code snippet* at lines 19 and 22.

9.7.2 New only flag and processor parameters

Let's now consider another common scenario where the 1-to-1 relation between two tables are not enough. You have implemented a `Processor` that is responsible to perform a gaussian filtering of your data and for this you have defined *gauss_sigma* as an `processor.ActiveParameter`. You run the pipeline for the first time and your *GaussFilter* is doing its job using *gauss_sigma* = 3, but then you realize that 3 is probably too much and you want to lower it down to say 2. You change the steering file and you re-run it and very likely nothing will happen. The reason is that if you have implemented the *new_only* filter in the `get_items()` as shown before, the output table is already containing all the filtered items from the input table.

A trick to force the re-processing is to delete the items manually from the database or one of its output files (if any and if you have included the `remove_widow_db_rows()` or `verify_checksum()`) but this is not really comfortable. The most elegant solution is to include a column in the output table to store the value of *gauss_sigma* and then adding a where condition in the query looking for the existing items.

Look at this query snippet example:

```
def get_items(self):
```

(continues on next page)

(continued from previous page)

```
    # add here all your checks, table creations, filter bindings

    if self.filter_register.new_only:
        query = (
            Image.select(OutputTable.input_id)
            .where(OutputTable.gauss_sigma == self.gauss_sigma)
            )
        existing = ~InputTable.input_id.in_([row.input_id for row in query])
    else:
        existing = True

    items = (InputTable.select(InputTable)
                .where(self.filter_register.filter_all())
                .where(existing)
                )

    return items
```

This approach is very handy, because it allows to link the entries in the database with the parameters set in the steering file, but it must be used with care, because changing a parameter will trigger the reprocessing of all entries while you might be thinking that this will only apply to the added items only.

9.7.3 Advanced *new_only* filtering with multiple column primary keys

Before moving on to the next section, we would like to show you another implementation tip where multiple column primary keys and *new_only* filtering work nice together. In the *basic usage* the new only flag the input and output tables were linked by a foreign key based on a single column primary key. Now let's see how we can handle the situation where the input table is the cross join of two tables with different primary keys (*image_id* from ProcessedImage and *method_id* from CalibrationMethod) and the output table CalibratedImage has the combined primary key (*image_id*, *method_id*). The models described here refers to the ones described *here*.

The idea is to use field combinations and for this you can rely on some helper functions defined in the *db_tools* module. Let's have a look at a possible implementation.

```
def get_items(self):

    # check if we want to process only new items
    if self.filter_register.new_only:
        # get a combined primary key because the output table (CalibratedImage) has
        # a multicolumn primary key
        existing_combo = combine_pk(CalibratedImage,
                                    alias_name = 'combo',
                                    join_str = ' x ')

        # execute a select on the output table for the combined field
        existing_entries = CalibratedImage.select(existing_combo).execute()

        # to find the existing entries in the input model, we need to combine the␣
↪fields
        # and compare them to the list of combined pks from the target model.
        # Pay attention to two things:
        #   1. In the where conditions you cannot have an alias, so the
        #       combine_fields is not aliased.
        #   2. The argument of the in_ must be a python list and not the pure select.
        existing = ~combine_fields(
            [ProcessedImage.image_id,
            CalibrationMethod.method_id],
```

(continues on next page)

(continued from previous page)

```
23              join_str = ' x '
24              ).in_([entry.combo for entry in existing_entries])
25      else:
26          existing = True
27
28      query = (
29          ProcessedImage.select(ProcessedImage, Image, Sample,
30                                Resolution, CalibrationMethod)
31          .join_from(ProcessedImage, Image, attr='_image')
32          .join_from(Image, Sample, on=(Image.sample_id == Sample.sample_id), attr='_
    ↪sample')
33          .join_from(Image, Resolution, on=(Image.resolution_id == Resolution.
    ↪resolution_id), attr='_resolution')
34          .join_from(ProcessedImage, CalibrationMethod, on=True, attr='_calibration_
    ↪method')
35          .where(existing)
36          .where(self.filter_register.filter_all())
37      )
38      return query
```

The code snippet is rather self-explanatory and well commented. The most important parts are the lines 7-9: they are the SQL equivalent of:

```
SELECT image_id | ' x ' | method_id FROM calibrated_image
```

so it will return one single column of text where each row is something like *1 x 1*, *2 x 1*….. You can change the joining string to whatever you like most, the only requirement is that it must be the same as the one used at line 23.

The next step is to build a query of all possible combinations of the two fields *ProcessedImage.image_id* and *CalibrationMethod.method_id*. This is obtained using the *combine_fields()* function. Pay attention to two small details:

1. The output of *combine_fields()* is meant to be used in the where condition and peewee does not supported aliased expression in where conditions.
2. The *in_* operator is expecting a python list as argument, so you need to transform the existing_entries query is a list with the combinations.

9.8 Why do I get an error about model binging?

It is possible that when using filters in your project, you will get an error at runtime mentioning that it was not possible to perform autobinding of the model or that the *ModelFilter* was no bound before it is use.

The reason for this error is very simple and the solution also straightforward. But let me fist explain what model binding means and how it is done. When you write your steering file and configure the filters, you identify the model class with a string in the TOML dictionary. The model is a class and the interpreter needs to connect (bind) the name you provide with the steering file with a Model class in order to perform the actual filters generation.

The current implementation of peewee does not provide a list or a dictionary of all the existing models in a database, but MAFw has a solution for this, the *ModelRegister* already introduced in the previous section *Automatic model registration*. If your Model inherits from *MAFwBaseModel*, then you are safe: the *processor filter* will ask the *model register* to provide the link between the model name and the actual class and your are good to go.

If your Models are not inheriting from *MAFwBaseModel* then you have to do the binding manually. It is not difficult, it is just a line of code to be added to the implementation of the get_items method just before preparing the query, Here is an example:

Listing 9.3: Manual filter - model binding

```
def get_items(self):

    # create your tables, if they still do not exist
    self.database.create_tables([Model1, Model2])

    # now bind the filters with the models
    self.filter_register.bind([Model1, Model2])

    # now continue the code as in the previous examples.
```

9.9 Filter migration from MAFw v1.4.0

If you were using MAFw in its early version v1.4, you might be find interesting reading this paragraph. Otherwise, skip it and go to the next chapter.

There were a lot of changes to the filter module going from v1.4 to v2.0, mainly in the implementation side but some also in the interface. So if you had steering files generated with version v1.4, they probably won't work directly with the newest version.

Here is a list of the important changes that you need to take care of:

1. **The GlobalFilter** is removed. In the old version, this was used to store the global new_only flag and, if implemented, some default filters to be applied to all processors in the pipeline. This functionality has become incompatible with the new system and has been removed. If you want to set the value of the new_only flag at the pipeline level, just use the new_only variable at the top level of the steering file and you can change it at Processor level using the *__new_only__* keyword.
2. **Filter replaced by __filter__**. In the old steering files, the TOML table containing the definition of the filters for each model was identified by the word *Filter* (with capitol F). This has now been replaced by the keyword *__filter__*, with double underscores at the beginning and at the end. This is to allow you to have a processor parameter named Filter and also it is more in line with the keyword style used for the logic statement, the new only tag and the conditional filters.
3. **Automatic binding**. In version v1.4, the filter binding with the model was under the responsibility of the user, meaning that it was your task to do the binding of each filter with its model before being able to generate the filtering expression. In version v2.0, thanks to the `ModelRegister` the binding process is happening automatically if your models inherit from `MAFwBaseModel`.
4. **Missing fields in a ModelFilter**. In version v1.4, if you had a field condition in a ModelFilter that was not matching any field in the bound model, the condition was silently ignored. If for example, you wanted to create a condition on a field named *my_image_id* and you wrote a condition line *image_id = 5*, the filter was not able to find the image_id field in the model and it was ignoring this statement. This behavior is incompatible with the new implementation. Now all conditions in a ModelFilter should match a field in the corresponding model. If the user provided a logic statement, not mentioning one or more conditions, they will silently ignored, otherwise the default behavior of AND'ing all of them will be applied.

9.10 What's next

The power of filters is incredible and you can control them via the steering file without having to change a single line in your code!

The next chapter is about the library of advanced processors that MAFw is sharing to simplify your job even further. In particular, we are sure you will like a lot our plotting processors!

CHAPTER 10

DISPLAY YOUR RESULTS

Scientists love to present their results graphically, because like a picture is worth a thousand words, a good plot can replace a lot of tables. The python scene is full of libraries for data visualization, from the most basic like matplotlib offering a low level interface to higher level tools like seaborn where the user can profit from predefined classes to display complex relationships among different variables in the dataset.

In a spirit of complete freedom, MAFw is not deciding for you what plotting or data frame library you should use for your analysis. For this reason, we have prepared a skeleton of the *`generic plotter processor`* that you can use as a template and implement it with your favorite libraries.

Nevertheless, if you are ok with the use of seaborn as a data visualization platform (using matplotlib under the hood for the low level plotting routines) and pandas and numpy for data handling, then MAFw can offer you a pre-cooked solution implemented in the *`SNSPlotter`*.

If you want to use *`SNSPlotter`*, then you need to install mafw including the *seaborn* optional feature, in this way:

```
c:\> pip install mafw[seaborn]
```

In the next section of this unit, we will assume that you are using the seaborn functionalities.

10.1 Add diagnostic information to your processors

Processors are responsible for performing simple analysis tasks; however, it is essential to verify and assess the accuracy of their output. We strongly recommend to include the generation of some diagnostic plots or storing some key results in a database table while implementing the *`process()`* method in order to verify the correct execution of the analysis task.

This aspect is totally in your hand and MAFw has little to offer; you will have to code your *`Processor`* so that it can create meaningful figures.

Warning

For this diagnostic output you can use the graphical library of your choice, but many of you will opt for *matplotlib* given its widespread use in the field. This library allows you to decide which backend to use from a long list of options including interactive and not interactive choices. If you let matplotlib decide, it will use one of the available backends according to a precedence list. In many cases, this will be *TkAgg* that is known to have issues when used in a non interactive way with frequently opening and closure of figure windows. Those issues will appear as randomly occurring RuntimeError involving TclAsync or main thread not in main loop. In such a case, the solution is to switch to a non-interactive backend that is normally absolutely ok because usually diagnostic output is simply saved on files and not directly displayed to you. The simplest non interactive backend is *Agg* and it is built-in in *matplotlib* so no need to install anything else.

You can force *matplotlib* to select a specific backend via the environment variable MPLBACKEND simply setting it to agg. This decision will affect all scripts (including mafw execution engine). If you want something that is specific for one or many processors of yours, you can add the following lines of code to your processor start method.

```
try:
    my_backend == 'agg'
    if plt.get_backend() != my_backend:
        plt.switch_backend(my_backend)
except ModuleNotFoundError:
    log.critical('%s is not a valid plt backend' % my_backend)
    raise
```

You can even have a processor whose only task is to turn on and off interactivity. The reason why this is not an option available in the base Processor class is that MAFw is graphics agnostic and unless you install the optional feature `seaborn` otherwise matplotlib is not automatically installed.

10.2 Generate high level graphs

Along with graphs aimed to verify that your processor was actually behaving as expected, you might have the need to have processors whose main task is indeed the generation of high level figures to be used for your publications or presentations starting from a data frame.

In this case, MAFw is providing you a powerful tool: the *generic plotter processor*.

The *SNSPlotter* is a member of the *processor_library* package and represents a specialized subclass of a *Processor*[1] with a predefined *process()* method.

When you are subclassing a *Processor*, you are normally required to implement some methods and the most important of all is clearly the *process()* one, where your analytical task is actually coded.

For a *SNSPlotter*, the *process method* is already implemented and it looks like the following code snipped:

```
def process(self) -> None:
    """
    Process method overload.

    In the case of a plotter subclass, the process method is already implemented and␣
↪the user should not overload
    it. On the contrary, the user must overload the other implementation methods␣
↪described in the general
    :class:`class description <.SNSPlotter>`.
    """
    if self.filter_register.new_only:
        if self.is_output_existing():
            return

    self.in_loop_customization()
    self.get_data_frame()
    self.patch_data_frame()
    self.slice_data_frame()
    self.group_and_aggregate_data_frame()
    if not self.is_data_frame_empty():
        self.plot()
        self.customize_plot()
        self.save()
        self.update_db()
```

[1] Actually *SNSPlotter* parent class is *GenericPlotter* that is a subclass of *Processor*. This intermediate inheritance step is needed to allow the user to define their preferred interface to graphical and data frame library.

In its basic form, all the methods included in the process have an empty implementation, and it is your task to make them doing something.

We can divide these methods into two blocks:

- **The data handling methods: where the data frame is acquired and prepared for the plotting step.**
    1. *get_data_frame()*
    2. *patch_data_frame()*
    3. *slice_data_frame()*
    4. *group_and_aggregate_data_frame()*
- **The plotting methods: where you will have to generate the figures from the data frame and save them.**
    1. *plot()*
    2. *customize_plot()*
    3. *save()*
    4. *update_db()*

10.2.1 Getting the data

The first thing to do when you want to plot some data is to have the data! The *SNSPlotter* has a *data_frame* attribute to store a pandas data frame exactly for this purpose. The data frame can be read in from a file or retrieved from a database table or even built directly in the processor itself. The *get_data_frame()* is the place where this is happening. It is important that you assign the data frame to *self.data_frame* attribute so that the following methods can operate on it.

The following three methods are giving you the possibility to add new columns based on the existing ones (*patch_data_frame()*), selecting a sub set of rows (*slice_data_frame()*) and grouping / aggregating the rows (*group_and_aggregate_data_frame()*).

Data frame patching

Use this method to add or modify data frame columns. One typical example is the application of unit of measurement conversion: you may want to store the data using the SI unit, but it is more convenient to display them using another more practical unit. Another interesting application is to replace numerical categorical values with textual one: for example you may want to add a column with the sample name to replace the column with the sample identification number. You can implement all these operations overloading the *patch_data_frame()* method. Below is an example of how to add an additional column that represents the exposure time of an image acquisition in hours, converted from the stored value in seconds.

```
self.data_frame['ExposureTimeH'] = self.data_frame['ExposureTime'] / 3600.
```

Slicing the data frame

Use this method to select a specific group of rows, corresponding to a certain value for a variable. You do not need to overload the *slice_data_frame()*, but just define the *slicing_dict* attribute and the basic implementation will do the magic. Have a look also at the documentation of the function *slice_data_frame()* from the *pandas_tools* module. In a few words, if you have a data frame with four columns named A, B, C and D and you want to select only the rows where column A is equal to 10, then set `self.slicing_dict = { 'A' : 10 }` and the *slice_data_frame()* method will do the job for you.

Group and aggregate the data frame

In a similar manner, you can group and aggregate the columns of the data frame by defining the *grouping columns* and *aggregation functions* attributes. Also for this method, have a look at the *group_and_aggregate_data_frame()* function from the *pandas_tools*. If, for example, you have a data frame containing two columns named SampleName and Signal and you want to get the average and standard deviation of the signal column for each of the different samples, then you can do:

```
self.grouping_columns = ['SampleName']
self.aggregation_functions = ['mean', 'std']
```

The *group_and_aggregate_data_frame()* will perform the grouping and aggregation and assign to the *self.data_frame* a modified data frame with three columns named SampleName, Signal_mean and Signal_std.

Per cycle customization

You may wonder, when and where in your code you should be setting the slicing and grouping / aggregating parameters. Those values can be 'statically' provided directly in the constructor of the generic plotter processor subclass, but you can also have them dynamically assigned at run time overloading the *in_loop_customization()* method.

In the implementation of this method you can also set other parameters relevant for your plotting.

10.2.2 Plotting the data

Now that you have retrieved and prepared your data frame, you can start the real plotting. First of all, just be sure that after all your modifications, the data frame is still containing some rows, otherwise, just move on to the next iteration.

The generation of the real plot is done in the implementation of the *plot()*. The *SNSPlotter* is meant to operate with seaborn so it is quite reasonable to anticipate that the results of the plotting will be assigned to a *facet grid attribute*, in order to allow the following methods to refer to the plotting output.

If your *plot()* implementation is not actually generating a facet grid, but something else, it is still ok. If you want to pass a static typing, then define a attribute in your *SNSPlotter* subclass with the proper typing; if you are not doing any static typing you can still use the facet grid attribute for storing your object.

Customising the plot

When producing high quality graphs, you want to have everything looking perfect. You can implement this method to set the figure and plot titles, the axis labels and to add specific annotations. If your *SNSPlotter* implementation is following a looping execution workflow, you can use the looping parameters if you need them. Theoretically speaking you could code this customisation directly in the plot method, but the reason for this split will become clear later on when explaining the *mixin approach*.

Save the plot

This is where your figure must be saved to a file. The very typical implementation of this method is the following:

```
def save(self) -> None:
    self.facet_grid.figure.savefig(self.output_filename)
    self.output_filename_list.append(self.output_filename)
```

This is assuming that your *SNSPlotter* subclass has an attribute (it can also be a processor parameter) named output_filename. The second statement is also very common: the *SNSPlotter* has a list type attribute to store the names of all generated files. This list, along with the cumulative checksum will be stored in the dedicated *PlotterOutput* standard table, to avoid the regeneration of identical plots. You do not have to worry about updating this standard table, this operation will be performed automatically by the private method *_update_plotter_db()*.

Update the db

Along with the *PlotterOutput* standard table, you may want to update also other objects in the database. For this purpose, you can implement the *update_db()* method.

10.2.3 Mixin approach

You might have noticed that the structure of the *process()* is a bit *over-granular*. The reason behind this fine distinction is due to the fact that the code can potentially be reused to a large extent thanks to a clever use of mixin classes.

Explanation

If you are not a seasoned programmer, you might not be familiar with the concept of mixins. Those are small classes implementing a specific subtask of a bigger class, finding extensive usage in developing interfaces. If you wish, you can have a general explanation following this webpage.

If you are aware of what a mixin is and how to use it, then just go to the *next section*.

In the case of the *SNSPlotter*, we have seen that its methods can be divided into two categories: the data retriever and the actual plotter. You can have several different data retriever options and also different plotting strategies, thus making the creation of a matrix of subclasses of the *SNSPlotter* covering all possible cases really unpractical.

The mixin approach allows to have a flexible subclass scheme and at the same time an optimal reuse of code units.

To make things even more evident, here is a logical scheme of what happens when you mix a *DataRetriever* mixin with a *SNSPlotter*.

Let us start from the definition of the subclass itself:

```
class MyPlotterSNS(MyDataRetrieverMixin, MyPlotMixin, SNSPlotter):
```

MyPlotterSNS is mixing two mixins, one *DataRetriever* and one *FigurePlotter*, with the *SNSPlotter*. The order **matters**: always put mixins before the base class, because this is affecting the method resolution order (MRO) that follows a depth-first left-to-right traversal.

`MyDataRetrieverMixin` being a subclass of *DataRetriever* is implementing these two methods *get_data_frame()* and *patch_data_frame()* that are also defined in the *SNSPlotter*.

During the execution of the *process()*, the `get_data_frame` method will be called and, thanks to the MRO, the MyDataRetrieverMixin will provide its implementation of such method. The same will apply for the `plot`, where the MyPlotMixin will kick in and provide its implementation of the method.

Mixins can come with other class parameters along with the ones shared with the base class. Those can be accessed and assigned using the standard dot notation on the instances of the mixed class, or they can also be defined in the mixed class constructor directly.

Data-retriever mixins

Let us start considering the data retrieval part. Very likely, you will get those data either from a database table or from a HDF5 file. If this is the case, then you do not need to code the get_data_frame method everytime, but you can use one of the available mixin to grab the data.

For the moment, the following data retriever mixins exist[2]:

- *HDFPdDataRetriever*.

  This data retriever is opening the provided HDF5 file name and obtaining the data frame identified by the provided key.

- *SQLPdDataRetriever*.

  This data retriever is performing a SQL query on a table (*table_name*) using as columns the provided *required_columns* list and fulfilling the *where_clause*. A valid database connection must exist for this retriever to work, it means that the processor with which this class is mixed must have a working database instance.

- *FromDatasetDataRetriever*.

[2] The extra *Pd* in the name of these classes stands for Pandas. In fact the user is free to select any other data frame library and define their own data retriever mixin.

This data retriever is provided mainly for test and debug purposes. It allows to obtain a data frame from the standard data set library of seaborn.

If you plan to retrieve data several times from a common source, you may consider to add your own mixin class. Just make it inheriting from the *DataRetriever* (or *PdDataRetriever* if you are using Pandas) class and follow one of the provided case as an example.

Here below is an example of a Plotter retrieving the data from a HDF file:

```
@single_loop
class HDFPlotterSNS(HDFDataRetriever, SNSPlotter):
    pass

hdf = HDFPlotterSNS(hdf_filename=your_input_file, key=your_key)
hdf.execute()
```

The hdf_filename and the key can be provided directly in the class constructor, but this might not doable if you are planning to execute the processor using the *mafw steering file* approach.

In this case, you can add two parameters to your processor that can be read from the steering file and follow this approach:

```
@single_loop
class HDFPlotterSNS(HDFDataRetriever, SNSPlotter):
    hdf_filename_from_steering = ActiveParameter('hdf_filename_from_steering',␣
↪default='my_datafile.h5')
    hdf_key_from_steering = ActiveParameter('hdf_key_from_steering', default='my_key')

    def start(self):
        super().start()
        # you can assign the parameter values to the mixin attributes in the start␣
↪method
        # but also in the in_loop_customization method if this is better suiting your␣
↪needs
        self.hdf_filename = self.hdf_filename_from_steering
        self.key = self.hdf_key_from_steering

# let us execute the processor from within the mafw runner.
```

Figure plot mixins

In a similar way, if what you want is to generate a seaborn figure level plot, then you do not need to define the plot method every time. Just mix the SNSPlotter with one of the four figure plotter mixins available:

- *RelPlot*: to plot relational plots.
- *DisPlot*: to plot distribution graphs.
- *CatPlot*: to produce categorical figures.
- *LMPlot*: to produce linear regression model figures.

From the coding point of it, the approach is identical to the one shown before for the data retrieving mixin.

```
@single_loop
@processor_depends_on_optional(module_name='pandas;seaborn')
class DataSetRelPlotPlotterSNS(FromDatasetDataRetriever, RelPlot, SNSPlotter):
    def __init__(self, output_png: str | Path | None = None, *args: Any, **kwargs:␣
↪Any) -> None:
        super().__init__(*args, **kwargs)
        self.output_png = Path(output_png) if output_png is not None else Path()
```

(continues on next page)

(continued from previous page)

```
    def save(self) -> None:
        self.facet_grid.figure.savefig(self.output_png)
        self.output_filename_list.append(self.output_png)

output_file = tmp_path / Path('relplot_from_dataset.png')

dsrp = DataSetRelPlotPlotterSNS(
    output_file, dataset_name='penguins', x='flipper_length_mm', y='bill_length_mm',␣
↪col='sex', hue='species'
)
dsrp.execute()
```

The code above will generate this plot:

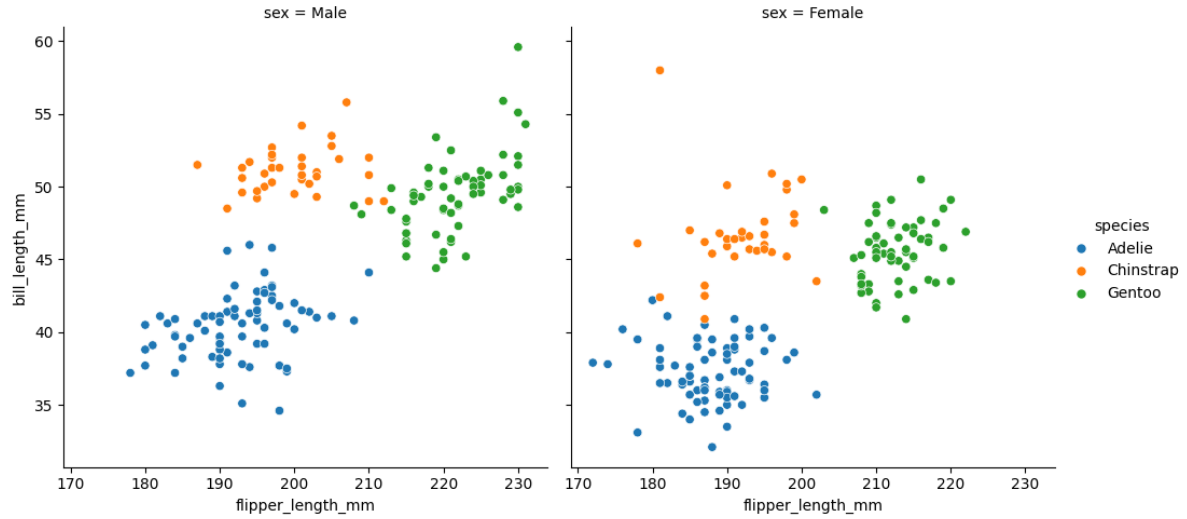


Fig. 10.1: This is the exemplary output of a SNSPlotter mixed with a data retriever and a relational plot mixin.

As you can see, the plot method is taken from the RelPlot mixin and the output can be further customized (axis labels and similar things) implementing the customize_plot method.

A large fraction of the parameters that can be passed to the seaborn figure level functions corresponds to attributes in the mixin class, but not all of them. For the missing ones, you can use the dictionary like parameter plot_kws, to have them passed to the underlying plotting function. See the documentation of the mixins for more details (*RelPlot*, *DisPlot* and *CatPlot*).

Note

Note for direct readers: If you have navigated directly to this practical example without reading the preceding documentation, please be aware that this implementation demonstrates advanced concepts and methodologies that are thoroughly explained in the earlier sections. While this example is designed to be self-contained, some technical details, and design decisions may not be immediately clear without the foundational knowledge provided in the main documentation. Should you encounter unfamiliar concepts or require deeper understanding of the library's architecture and principles, we recommend referring back to the relevant sections of the complete documentation.

CHAPTER 11

A STEP BY STEP TUTORIAL FOR A REAL EXPERIMENTAL CASE STUDY

Congratulations! You have decided to give a try to MAFw for your next analysis, you have read all the documentation provided, but you got overwhelmed by information and now you do not know from where to start.

Do not worry, we have a simple, but nevertheless complete example that will guide you through each and every steps of your analysis.

Before start typing code, let's introduce the experimental scenario.

11.1 The experimental scenario

Let's imagine, that we have a measurement setup integrating the amount of UV radiation reaching a sensor in a give amount of time.

The experimental data acquisition (DAQ) system is saving one file for each exposure containing the value read by the sensor. The DAQ is encoding the duration of the acquisition in the file name and let's assume we acquired 25 different exposures, starting from 0 up to 24 hours. The experimental procedure is repeated for three different detectors having different type of response and the final goal of our experiment is to compare their performance.

It is an ultra simplified experimental case and you can easily make it as complex as you wish, just by adding other variables (different UV lamps, detector operating conditions, background illumination...). Nevertheless this simple scenario can be straightforward expanded to any real experimental campaign.

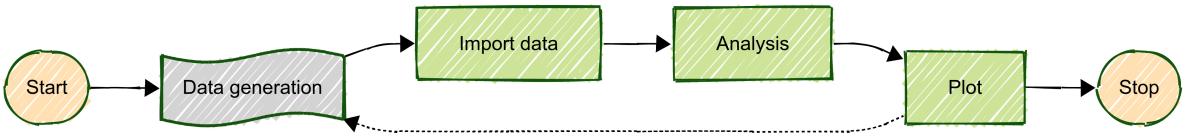


Fig. 11.1: The simplified pipeline for the tutorial example.

11.1.1 Task 0. Generating the data

This is not really an analysis task, rather the real acquisition of the data, that's why it is represented with a different color in schema above. Nevertheless it is important to include that in our planning, because new data might be generated during or after some data have been already analyzed. In this case, it is important that our analysis pipelines will be able to process only the new (or modified) data, without wasting time and resources re-doing what has been done already. This is the reason why there is a dashed line looping back from the plot step to the data generation. Since this is a simulated experiment, instead of collecting real data, we will use a `base.Processor` to generate some synthetic data.

11.1.2 Task 1. Building your data bank

From a conceptual point of view, the first thing you should do when using MAFw is to import all your data (in this case the raw data files) into a relational database. You do not need to store the content of the file in the database, otherwise it will soon explode in size, you can simply insert the full path from where the file can be retrieved and its checksum, so that we can keep an eye on possible modifications.

11.1.3 Task 2. Do the analysis

In the specific scenario the analysis is very easy. Each file contains only one number, so there is very little to be done, but your case can be as complicated as needed. We will open each file, read the content and then put it in a second database table containing the results of the analysis of each file. In your real life experiment, this stage can contain several processors generating intermediate results stored as well in the database.

11.1.4 Task 3. Prepare a relation plot

Using the data stored in the database, we can generate a plot representing the integral flux versus the exposure time for the three different detectors using a relation plot.

11.2 The 'code'

In the previous section, we have defined what we want to achieve with our analysis (it is always a good idea to have a plan before start coding!). Now we are ready to start with setting up the project containing the required processors to achieve the analytical goal described above.

If you want to use MAFw plugin mechanism, then you need to build your project as a proper python package. Let's start then with the project specification contained in the `pyproject.toml` file.

```
[build-system]
requires = ["hatchling"]
build-backend = "hatchling.build"

[project]
name = "plug_test"
dynamic = ["version"]
description = 'A MAFw processor library plugin'
readme = "README.md"
requires-python = ">=3.11"
license = "EUPL-1.2"
authors = [
  { name = "Antonio Bulgheroni", email = "antonio.bulgheroni@ec.europa.eu" },
]
classifiers = [
  "Development Status :: 4 - Beta",
  "Programming Language :: Python",
  "Programming Language :: Python :: 3.11",
  "Programming Language :: Python :: 3.12",
  "Programming Language :: Python :: 3.13",
  "Programming Language :: Python :: Implementation :: CPython",
  "Programming Language :: Python :: Implementation :: PyPy",
]

dependencies = ['mafw', 'pluggy']

[project.urls]
Documentation = "https://github.com/..."
Issues = "https://github.com/..."
Source = "https://github.com/."
```

(continues on next page)

(continued from previous page)

```

[project.entry-points.mafw]
plug_test = 'plug.plugins'

[tool.hatch.version]
path = "src/plug/__about__.py"

[tool.hatch.build.targets.wheel]
packages = ['src/plug']
```

The specificity of this configuration file is in the highlighted lines: we define an entry point where your processors are made available to MAFw.

Now before start creating the rest of the project, prepare the directory structure according to the python packaging specification. Here below is the expected directory structure.

```
plug
├── README.md
├── pyproject.toml
└── src
    └── plug
        ├── __about__.py
        ├── db_model.py
        ├── plug_processor.py
        └── plugins.py
```

You can divide the code of your analysis in as many python modules as you wish, but for this simple project we will keep all processors in one single module *plug_processor.py*. We will use a second module for the definition of the database model classes (*db_model.py*). Additionally we will have to include a *plugins.py* module (this is the one declared in the pyproject.toml entry point definition) where we will list the processors to be exported along with our additional standard tables.

11.2.1 The database definition

Let's move to the database definition.

Our database will contain tables corresponding to the three model classes: InputFile and Data, and one helper table for the detectors along with all the *standard tables* that are automatically created by MAFw.

Before analysing the code let's visualize the database with the ERD.

The InputFile is the model where we will be storing all the data files that are generated in our experiment while the Data model is where we will be storing the results of the analysis processor, in our specific case, the value contained in the input file.

The rows of these two models are linked by a 1-to-1 relation defined by the primary key.

Remember that is always a good idea to add a checksum field every time you have a filename field, so that we can check if the file has changed or not.

The InputFile model is also linked with the Detector model to be sure that only known detectors are added to the analysis.

Let's have a look at the way we have defined the three models.

```
class Detector(StandardTable):
    detector_id = AutoField(primary_key=True, help_text='Primary key for the detector␣
↪table')
    name = TextField(help_text='The name of the detector')
    description = TextField(help_text='A longer description for the detector')

```

(continues on next page)

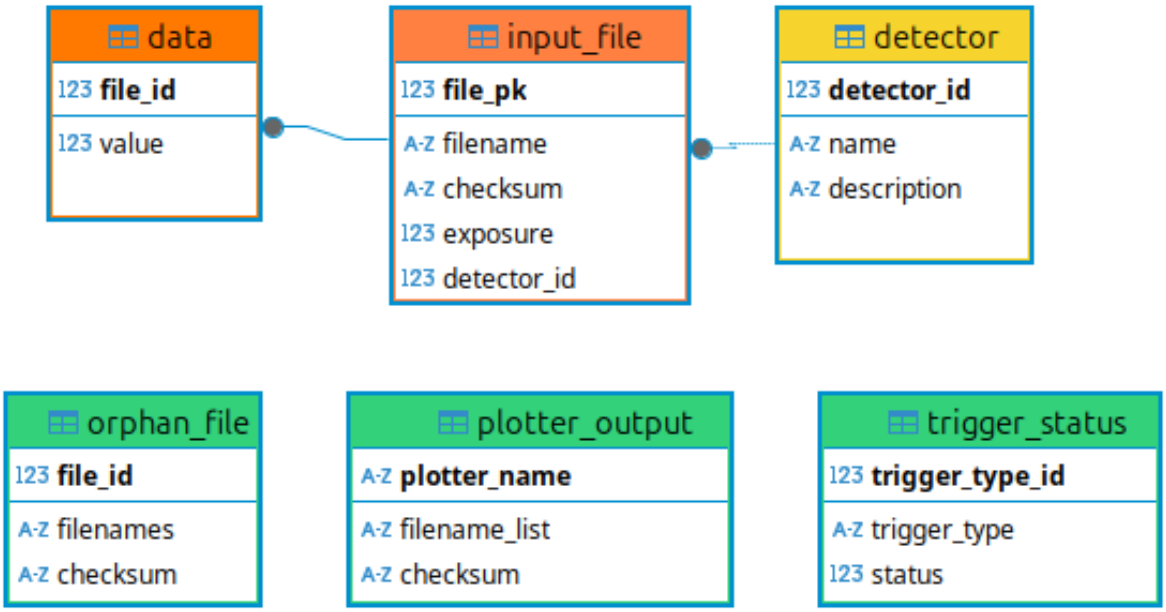


Fig. 11.2: The schematic representation of our database. The standard tables, automatically created are in green. The detector table (yellow) is exported as a standard table and its content is automatically restored every time mafw is executed.

(continued from previous page)

```
    @classmethod
    def init(cls) -> None:
        data = [
            dict(detector_id=1, name='Normal', description='Standard detector'),
            dict(detector_id=2, name='HighGain', description='High gain detector'),
            dict(detector_id=3, name='NoDark', description='Low dark current detector
↪'),
        ]

        cls.insert_many(data).on_conflict(
            conflict_target=[cls.detector_id],
            update={'name': SQL('EXCLUDED.name'), 'description': SQL('EXCLUDED.
↪description')},
        ).execute()
```

The detector table is derived from the StandardTable, because we want the possibility to initialize the content of this table every time the application is executed. This is obtained in the init method. The use of the on_conflict clause assure that the three detectors are for sure present in the table with the value given in the data object. This means that if the user manually changes the name of one of these three detectors, the next time the application is executed, the original name will be restored.

```
class InputFile(MAFwBaseModel):
    @classmethod
    def triggers(cls) -> list[Trigger]:
        update_file_trigger = Trigger(
            trigger_name='input_file_after_update',
            trigger_type=(TriggerWhen.After, TriggerAction.Update),
            source_table=cls,
            safe=True,
            for_each_row=True,
        )
```

(continues on next page)

(continued from previous page)

```
        update_file_trigger.add_when(or_('NEW.exposure != OLD.exposure', 'NEW.checksum != OLD.checksum'))
        update_file_trigger.add_sql('DELETE FROM data WHERE file_pk = OLD.file_pk;')

        return [update_file_trigger]

    file_pk = AutoField(primary_key=True, help_text='Primary key for the input file table')
    filename = FileNameField(unique=True, checksum_field='checksum', help_text='The filename of the element')
    checksum = FileChecksumField(help_text='The checksum of the element file')
    exposure = FloatField(help_text='The duration of the exposure in h')
    detector = ForeignKeyField(
        Detector, Detector.detector_id, on_delete='CASCADE', backref='detector', column_name='detector_id'
    )
```

The InputFile has five columns, one of which is a foreign key linking it to the Detector model. Note that we have used the FileNameField and FileChecksumField to take advantage of the *verify_checksum()* function. InputFile has a trigger that is executed after each update that is changing either the exposure or the file content (checksum). When one of these conditions is verified, then the corresponding row in the Data file will be removed, because we want to force the reprocessing of this file since it has changed. A similar trigger on delete is actually not needed because the Data model is linked to this model with an on_delete cascade option.

```
class Data(MAFwBaseModel):
    @classmethod
    def triggers(cls) -> list[Trigger]:
        delete_plotter_sql = 'DELETE FROM plotter_output WHERE plotter_name = "PlugPlotter";'

        insert_data_trigger = Trigger(
            trigger_name='data_after_insert',
            trigger_type=(TriggerWhen.After, TriggerAction.Insert),
            source_table=cls,
            safe=True,
            for_each_row=False,
        )
        insert_data_trigger.add_sql(delete_plotter_sql)

        update_data_trigger = Trigger(
            trigger_name='data_after_update',
            trigger_type=(TriggerWhen.After, TriggerAction.Update),
            source_table=cls,
            safe=True,
            for_each_row=False,
        )
        update_data_trigger.add_when('NEW.value != OLD.value')
        update_data_trigger.add_sql(delete_plotter_sql)

        delete_data_trigger = Trigger(
            trigger_name='data_after_delete',
            trigger_type=(TriggerWhen.After, TriggerAction.Delete),
            source_table=cls,
            safe=True,
            for_each_row=False,
        )
```

(continues on next page)

(continued from previous page)

```
        delete_data_trigger.add_sql(delete_plotter_sql)

        return [insert_data_trigger, delete_data_trigger, update_data_trigger]

    file_pk = ForeignKeyField(InputFile, on_delete='cascade', backref='file', primary_
↪key=True, column_name='file_id')
    value = FloatField(help_text='The result of the measurement')
```

The Data model has only two columns, one foreign key linking to the InputFile and one with the value calculated by the Analysis processor. It is featuring three triggers executed on INSERT, UPDATE and DELETE. In all these cases, we want to be sure that the output of the PlugPlotter is removed so that a new one is generated. Keep in mind that when a row is removed from the PlotterOutput model, the corresponding files are automatically added to the OrphanFile model for removal from the filesystem the next time a processor is executed.

Via the use of the foreign key, it is possible to associate a detector and the exposure for this specific value.

11.2.2 The processor library

Let's now prepare one processor for each of the tasks that we have identified in our planning. We will create a processor also for the data generation.

GenerateDataFiles

This processor will accomplish Task 0 and it is very simple. It will generate a given number of files containing one single number calculated given the exposure, the slope and the intercept. The detector parameter is used to differentiate the output file name. As you see here below, the code is very simple.

```
class GenerateDataFiles(Processor):
    n_files = ActiveParameter('n_files', default=25, help_doc='The number of 1-h␣
↪increasing exposure')
    output_path = ActiveParameter(
        'output_path', default=Path.cwd(), help_doc='The path where the data files␣
↪are stored.'
    )
    slope = ActiveParameter(
        'slope', default=1.0, help_doc='The multiplication constant for the data␣
↪stored in the files.'
    )
    intercept = ActiveParameter(
        'intercept', default=5.0, help_doc='The additive constant for the data stored␣
↪in the files.'
    )
    detector = ActiveParameter(
        'detector', default=1, help_doc='The detector id being used. See the detector␣
↪table for more info.'
    )

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.n_digits = len(str(self.n_files))

    def start(self) -> None:
        super().start()
        self.output_path.mkdir(parents=True, exist_ok=True)

    def get_items(self) -> Collection[Any]:
        return list(range(self.n_files))
```

(continues on next page)

(continued from previous page)

```
    def process(self) -> None:
        current_filename = self.output_path / f'rawfile_exp{self.i_item:0{self.n_
↪digits}}_det{self.detector}.dat'
        value = self.i_item * self.slope + self.intercept
        with open(current_filename, 'w') as f:
            f.write(str(value))

    def format_progress_message(self) -> None:
        self.progress_message = f'Generating exposure {self.i_item} for detector
↪{self.detector}'
```

In order to generate the different detectors, you run the same processor with different values for the parameters.

PlugImporter

This processor will accomplish Task 1, i.e. import the raw data file into our database. This processor is inheriting from the basic *Importer* so that we can use the functionalities of the *FilenameParser*.

```
@database_required
class PlugImporter(Importer):
    def __init__(self, *args: Any, **kwargs: Any) -> None:
        super().__init__(*args, **kwargs)
        self._data_list: list[dict[str, Any]] = []

    def get_items(self) -> Collection[Any]:
        pattern = '**/*dat' if self.recursive else '*dat'
        input_folder_path = Path(self.input_folder)

        file_list = [file for file in input_folder_path.glob(pattern) if file.is_
↪file()]

        # verify the checksum of the elements in the input table. if they are not up␣
↪to date, then remove the row.
        verify_checksum(InputFile)

        if self.filter_register.new_only:
            # get the filenames that are already present in the input table
            existing_rows = InputFile.select(InputFile.filename).namedtuples()
            # create a set with the filenames
            existing_files = {row.filename for row in existing_rows}
            # filter out the file list from filenames that are already in the␣
↪database.
            file_list = [file for file in file_list if file not in existing_files]

        return file_list

    def process(self) -> None:
        try:
            new_file = {}
            self._filename_parser.interpret(self.item.name)
            new_file['filename'] = self.item
            new_file['checksum'] = self.item
            new_file['exposure'] = self._filename_parser.get_element_value('exposure')
            new_file['detector'] = self._filename_parser.get_element_value('detector')
            self._data_list.append(new_file)
        except ParsingError:
```

(continues on next page)

(continued from previous page)

```
            log.critical('Problem parsing %s' % self.item.name)
            self.looping_status = LoopingStatus.Skip

    def finish(self) -> None:
        InputFile.insert_many(self._data_list).on_conflict_replace(replace=True).
↪execute()
        super().finish()
```

The get_items is using the *verify_checksum()* to verify that the table is still actual and we apply the filter to be sure to process only new or modified files. The process and finish are very standard. In this specific case, we preferred to add all the relevant information in a list and insert them all in one single call to the database. But also the opposite approach (no storing, multiple insert) is possible.

Analyser

This processor will accomplish Task 2, i.e. the analysis of the files. In our case, we just need to open the file, read the value and put it in the database.

```
@database_required
class Analyser(Processor):
    def get_items(self) -> Collection[Any]:
        self.filter_register.bind_all([InputFile])

        if self.filter_register.new_only:
            existing_entries = Data.select(Data.file_pk).execute()
            existing = ~InputFile.file_pk.in_([i.file_pk for i in existing_entries])
        else:
            existing = True

        query = InputFile.join_dependent_models().where(self.filter_register.filter_
↪all()).where(existing)

        return query

    def process(self) -> None:
        with open(self.item.filename) as fp:
            value = float(fp.read())

        Data.create(file_pk=self.item.file_pk, value=value)

    def format_progress_message(self) -> None:
        self.progress_message = f'Analysing {self.item.filename.name}'
```

Also in this case, the generation of the item list is done keeping in mind the possible *filters* the user is applying in the steering file. In this respect the use of the *join_dependent_models()* is very practical because it automatically generates a select joining all tables for which a foreign key relationship with the input one (InputFile in this case) exists. You do not have to worry to write the various multiple joins, MAFw will take care of doing it for you. In the process, we are inserting the data directly to the database, so we will have one query for each item.

PlugPlotter

This processor will accomplish the last task, i.e. the generation of a relation plot where the performance of the three detectors is compared.

```
@database_required
@processor_depends_on_optional(module_name='seaborn')
@single_loop
```

(continues on next page)

(continued from previous page)

```
class PlugPlotter(SQLPdDataRetriever, RelPlot, SNSPlotter):
    new_defaults = {
        'output_folder': Path.cwd(),
    }

    def __init__(self, *args, **kwargs):
        super().__init__(
            *args,
            table_name='data_view',
            required_cols=['exposure', 'value', 'detector_name'],
            x='exposure',
            y='value',
            hue='detector_name',
            facet_kws=dict(legend_out=False, despine=False),
            **kwargs,
        )

    def start(self) -> None:
        super().start()

        sql = """
        CREATE TEMP VIEW IF NOT EXISTS data_view AS
        SELECT
            file_id, detector.detector_id, detector.name as detector_name, exposure,␣
↪value
        FROM
            data
            JOIN input_file ON data.file_id = input_file.file_pk
            JOIN detector USING (detector_id)
        ORDER BY
            detector.detector_id ASC,
            input_file.exposure ASC
            ;
        """
        self.database.execute_sql(sql)

    def customize_plot(self):
        self.facet_grid.set_axis_labels('Exposure', 'Value')
        self.facet_grid.figure.subplots_adjust(top=0.9)
        self.facet_grid.figure.suptitle('Data analysis results')
        self.facet_grid._legend.set_title('Detector type')

    def save(self) -> None:
        output_plot_path = self.output_folder / 'output.png'

        self.facet_grid.figure.savefig(output_plot_path)
        self.output_filename_list.append(output_plot_path)
```

This processor is a mixture of *SQLPdDataRetriever*, *RelPlot* and *SNSPlotter*. The *SNSPlotter* has already some parameters and with the *new_defaults* dictionary we *over ride* value of the output_folder to point to the current folder.

Looking at the init method, you might notice a strange thing, the table_name variable is set to *data_view*, that does not corresponding to any of our tables. The reason for this strangeness is quickly explained.

The *SQLPdDataRetriever* is generating a pandas Dataframe from a SQL query. In our database the data table contains only two columns: the file reference and the measured value, so we have no direct access to the exposure nor to the detector. To get these other fields we need to join the data table with the input_file and the detector ones.

The solution for this problem is the creation of a temporary view containing this join query. Have a look at the start method. This view will be deleted as soon as the connection will be closed.

11.2.3 The plugin module

Everything is ready, we just have to make MAFw aware of our processors and our standard tables. We are missing just a few lines of code in the plugins module

```
import mafw
from mafw.lazy_import import LazyImportProcessor, ProcessorClassProtocol


@mafw.mafw_hookimpl
def register_processors() -> list[ProcessorClassProtocol]:
    return [
        LazyImportProcessor('plug.plug_processor', 'GenerateDataFiles'),
        LazyImportProcessor('plug.plug_processor', 'PlugImporter'),
        LazyImportProcessor('plug.plug_processor', 'Analyser'),
        LazyImportProcessor('plug.plug_processor', 'PlugPlotter'),
    ]


@mafw.mafw_hookimpl
def register_db_model_modules() -> list[str]:
    return ['plug.db_model']
```

The code is self-explaining. We need to invoke the processor hooks and return the list of processors. Instead of passing the real processor, we will use the *processors proxies*, so that we can defer the import of the processor modules when and if needed.

11.3 Run the code!

We are done with coding and we are ready to run our analysis.

First thing, we need to install our package in a separated python environment.

Windows

```
c:\> python -m venv my_venv
c:\> cd my_venv
c:\my_venv> bin\activate
(my_venv) c:\my_venv> pip install -e c:\path\to\plug
```

linux & MacOS

```
$ python -m venv my_venv
$ source my_venv/bin/activate
(my_venv) $ pip install -e /path/to/plug
```

Now verify that the installation was successful. If you run mafw list command you should get the list of all available processors including the three that you have created.

Windows

```
(my_venv) c:\my_venv> mafw list
```

linux & MacOS

```
(my_venv) $ mafw list
```

One last step, before running the analysis. We need to make the two steering files, one for the generation of the synthetic data and one for the real analysis and also the configuration file for the importer.

File generation

```toml
# file: generate-file.toml
processors_to_run = ["GenerateDataFiles#1", "GenerateDataFiles#2", "GenerateDataFiles
↪#3"]

# customise the name of the analysis
analysis_name = "integration-test-p1"
analysis_description = """Generating data files"""
available_processors = ["GenerateDataFiles"]

[GenerateDataFiles]
n_files = 25 # The number of files being generated.
output_path = "/tmp/full-int/data" # The path where the data files are stored.
slope = 1.0 # The multiplication constant for the data stored in the files.
detector = 1 # The detector id being used. See the detector table for more info.

["GenerateDataFiles#1"]
intercept = 5.0 # The additive constant for the data stored in the files.
slope = 1.0 # The multiplication constant for the data stored in the files.
detector = 1 # The detector id being used. See the detector table for more info.

["GenerateDataFiles#2"]
intercept = 15.0 # The additive constant for the data stored in the files.
slope = 5.0 # The multiplication constant for the data stored in the files.
detector = 2 # The detector id being used. See the detector table for more info.

["GenerateDataFiles#3"]
intercept = 0.1 # The additive constant for the data stored in the files.
slope = 0.2 # The multiplication constant for the data stored in the files.
detector = 3 # The detector id being used. See the detector table for more info.

[UserInterface] # Specify UI options
interface = "rich" # Default "rich", backup "console"
```

Analysis

```toml
# file: analysis.toml
processors_to_run = ["TableCreator","PlugImporter","Analyser", "PlugPlotter"]

# customise the name of the analysis
analysis_name = "integration-test-p2"
analysis_description = """Analysing data"""
available_processors = ["PlugImporter", "Analyser", "PlugPlotter"]
new_only = true

[DBConfiguration]
URL = "sqlite:////tmp/full-int/plug.db" # Change the protocol depending on the DB␣
↪type. Update this file to the path of your DB.
```

(continues on next page)

(continued from previous page)

```
[DBConfiguration.parameters.sqlite]

[DBConfiguration.parameters.sqlite.pragmas] # Leave these default values, unless you␣
↪know what you are doing!
journal_mode = "wal"
cache_size = -64000
foreign_keys = 1
synchronous = 0

[TableCreator] # Processor to create all tables in the database.
apply_only_to_prefix = [] # Create only tables whose name start with the provided␣
↪prefixes.
force_recreate = false

[PlugImporter]
input_folder = "/tmp/full-int/raw_data" # The input folder from where the images have␣
↪to be imported.
parser_configuration = "/tmp/full-int/importer_config.toml" # The path to the TOML␣
↪file with the filename parser configuration
recursive = true

[Analyser]

[PlugPlotter]
output_plot_path = "/tmp/full-int/output.png" # The filename of the output plot

[UserInterface] # Specify UI options
interface = "rich" # Default "rich", backup "console"
```

Importer configuration

```
# file: importer_config.toml
elements = ['exposure', 'detector']

[exposure]
regexp = '[_-]*exp(?P<exposure>\d+\.*\d*)[_-]*'
type='float'

[detector]
regexp = '[_-]*det(?P<detector>\d+)[_-]*'
type='int'
```

Adapt the steering files, in particular the paths and you are ready to run! In the analysis TOML file, you will also find the section concerning the database; for this simple case we used a SQLite single file DB, but whatever *other DB* would work exactly in the same way.

Windows

```
(my_venv) c:\my_venv> mafw run generate-file.toml
```

linux & MacOS

```
(my_venv) $ mafw run generate-file.toml
```

This will run the GenerateDataFiles processor three times, as specified in the three replicas, one for each detector. In the steering file, you can see that there are base settings specified in the base GenerateDataFiles table, plus

some specific values for the three replicas. If you need are refresh on processor replicas, go back *here*. The three processor replicas will generate all our input data and we are ready to start the data analysis.

Windows

```
(my_venv) c:\my_venv> mafw run analysis.toml
```

linux & MacOS

```
(my_venv) $ mafw run analysis.toml
```

And here comes the magic! The three processors will be executed one after the other, the database is created and filled with all the provided data and the comparison plot is generated (default file name output.png).

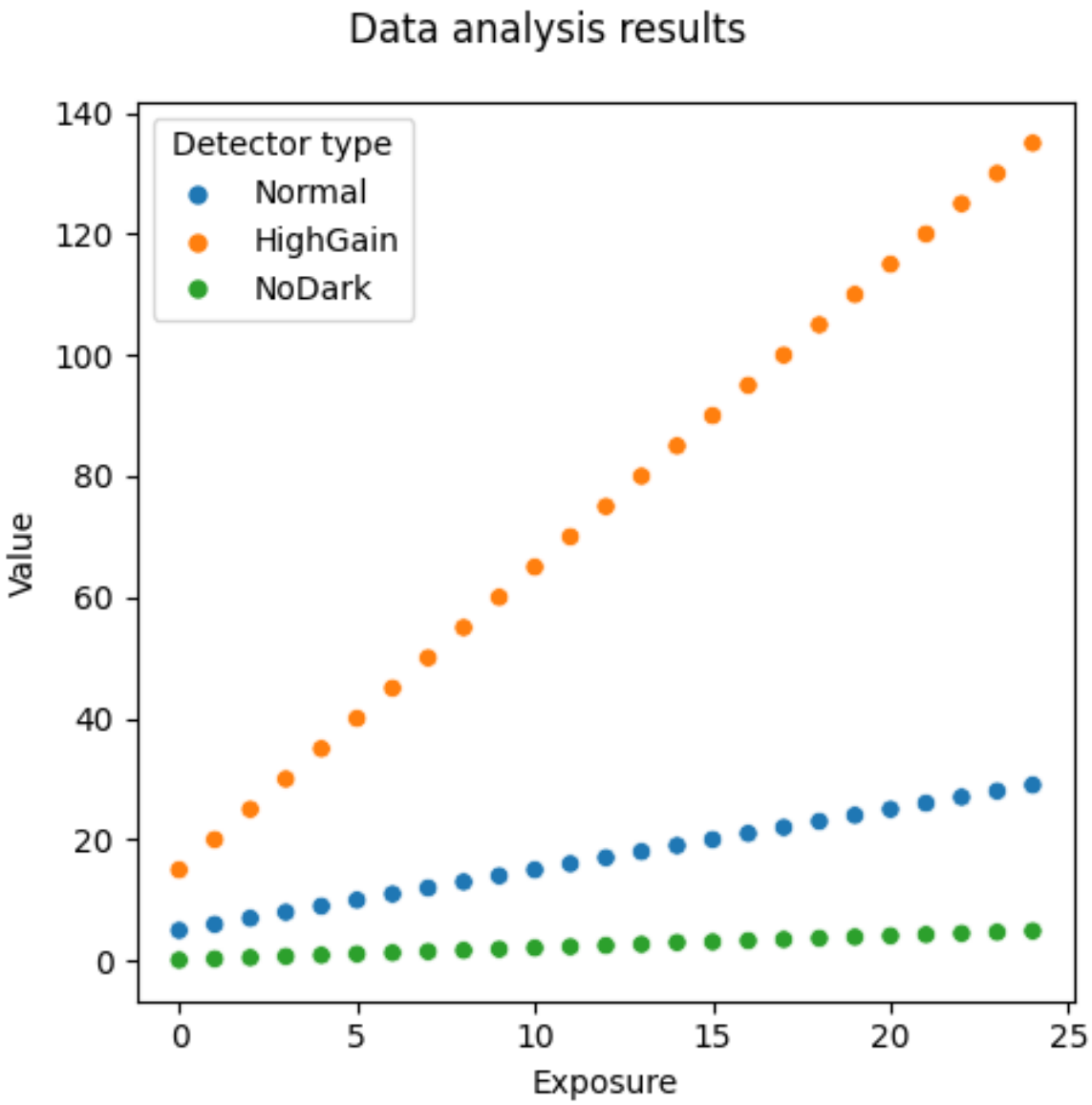


Fig. 11.3: The comparison plot of the three detectors.

This is just the beginning, now you can try all the benefits to use a clever database to drive your analysis pipeline. Try, for example, to remove one file and re-run the analysis, you will see a warning message informing you that a

file was not found and that the corresponding row in the database has been removed as well. The rest of the analysis will remain the same, but the output plot will be regenerated with a missing point.

Try to manually modify the content of a file and re-run the analysis. The *verify_checksum()* will immediately detect the change and force the re-analysis of that file and the generation of the output plot.

Try to rename one file changing the exposure value. You will see that mafw will detect that one file is gone missing in action, but a new one has been found. The output file will be update.

Try to repeat the analysis without any change and mafw won't do anything! Try to delete the output plot and during the next run mafw will regenerate it.

You can also play with the database. Open it in DBeaver (be sure that the foreign_check is enforced) and remove one line from the input_file table. Run the analysis and you will see that the output plot file is immediately removed because it is no more actual and a new one is generated at the end of chain.

It's not magic even if it really looks like so, it's just a powerful library for scientists written by scientists!

CHAPTER 12

STEERING FILES

This chapter describes how to create and manage steering files in MAFw. Before reading this chapter be sure to be familiar with the basic concept of MAFw, like processors, parameters and filters, otherwise you won't be able to understand what we are talking about. Steering files are TOML documents that define the analysis pipeline, including global variables, database connection settings, user interface preferences, and the list of processors to be executed.

12.1 Steering GUI User Guide

Writing steering files by hand can be a daunting task. While TOML is a human-readable format, it is easy to make a typo in a processor name, forget a mandatory parameter, or create a circular dependency between groups. To make your life easier, MAFw provides a graphical tool to generate and edit these files in a user-friendly way.

12.1.1 Why use the Steering GUI?

- **Error prevention**: No more worrying about TOML syntax or misspelled keys. The GUI ensures that the file is always well-formatted.
- **Automatic validation**: As you build your pipeline, the GUI checks for common mistakes like missing references or cyclic group definitions.
- **Schema alignment**: The GUI knows what parameters each processor expects. It automatically shows the correct editors for each field, so you don't have to look up the documentation every time.

12.1.2 Installation

The Steering GUI is not installed by default because it depends on PySide6, which is quite a heavy library and you do not need the steering GUI on a headless computational container. If you want to use it, you need to install it explicitly.

You can install it using pip:

```
pip install mafw[steering-gui]
```

Or, if you are developing MAFw and using `hatch`:

```
hatch run optional:steering-gui
```

> **Note**
>
> You might notice that the test coverage for the GUI is not as high as for the rest of the framework. This is intentional: we focus our testing efforts on the scientific "business logic" to ensure your results are always correct. Testing graphical interfaces is notoriously difficult and would significantly slow down our development and continuous integration.

Once installed, you can run the GUI either from the command line in your virtual environment with *steering-gui* or you can make a link to the binary file on your desktop and launch it with a double click.

12.1.3 General Operation

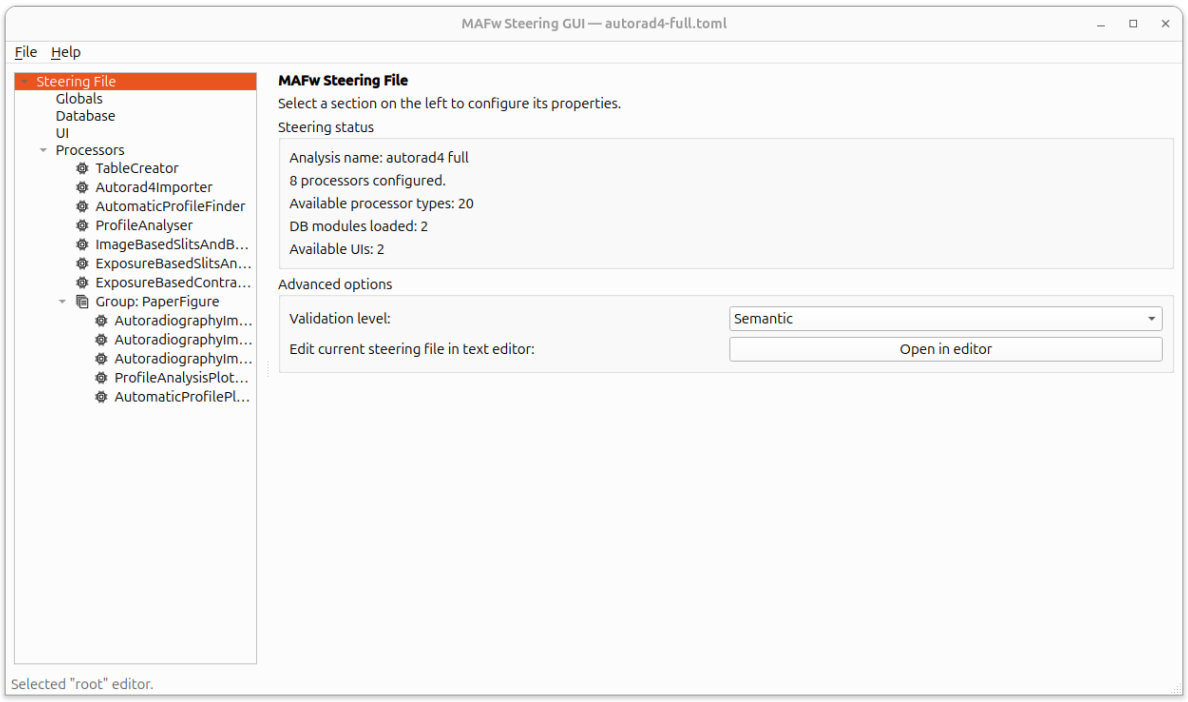
When you launch the Steering GUI, you are greeted with a traditional desktop layout.

- **File Menu**: Use this to create a **New** steering file, **Open** an existing one, or **Save** your progress.
- **Main Window: The window is split into two parts:**
  - **Pipeline Tree (Left)**: Shows the structure of your steering file, including Globals, Database, UI, and all your Processors and Groups.
  - **Editor Area (Right)**: Changes depending on what you have selected in the tree.
- **Context Menus**: Almost everything in the GUI has a right-click menu! If you are stuck or want to see what else you can do (like adding a new processor or removing a filter), just try right-clicking on an item.

12.1.4 The Editors

Metadata Editor

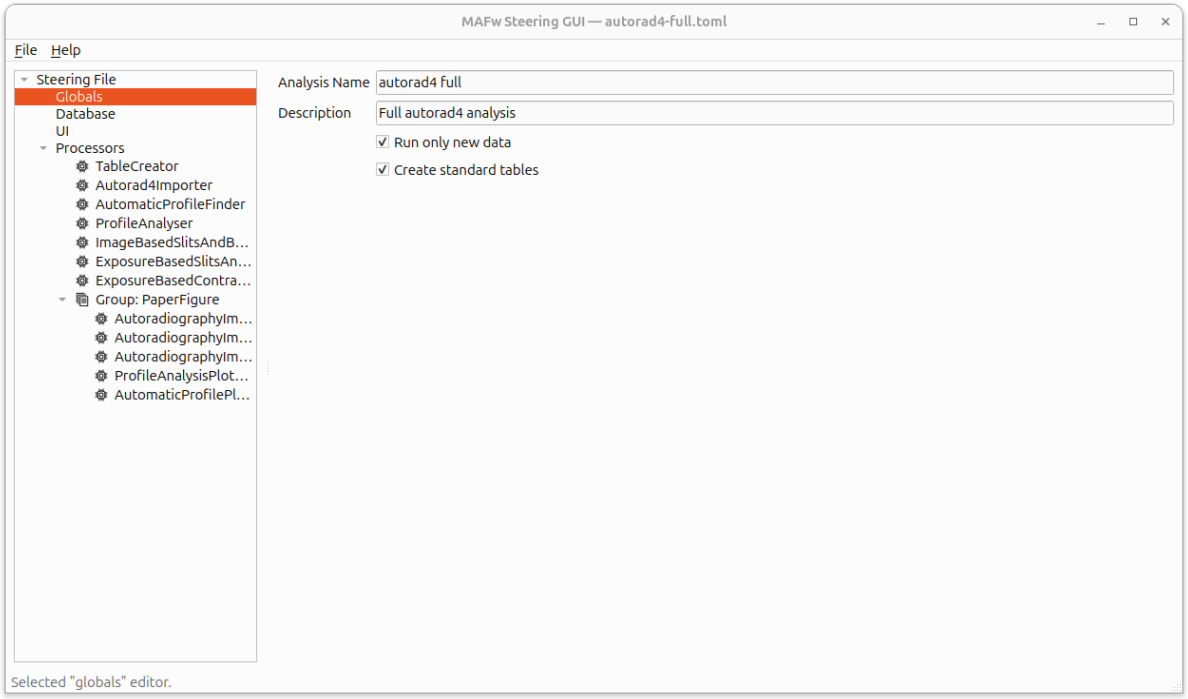
When you select the top-level item in the tree, you'll see a summary of your analysis. This includes the analysis name and a quick count of how many processors and groups you have configured.



If you ever feel the need to see the raw TOML or make a quick manual change, there is a **Open in editor** button that will open the current state of the file in a custom text editor with TOML syntax highlighting and live validation. It won't be as intuitive as with the GUI, but at least you won't make stupid typo mistakes.

Globals Editor

Here you can set the analysis name and description along with globally valid switches for the processing of *new_only* data and the creation of standard database tables.



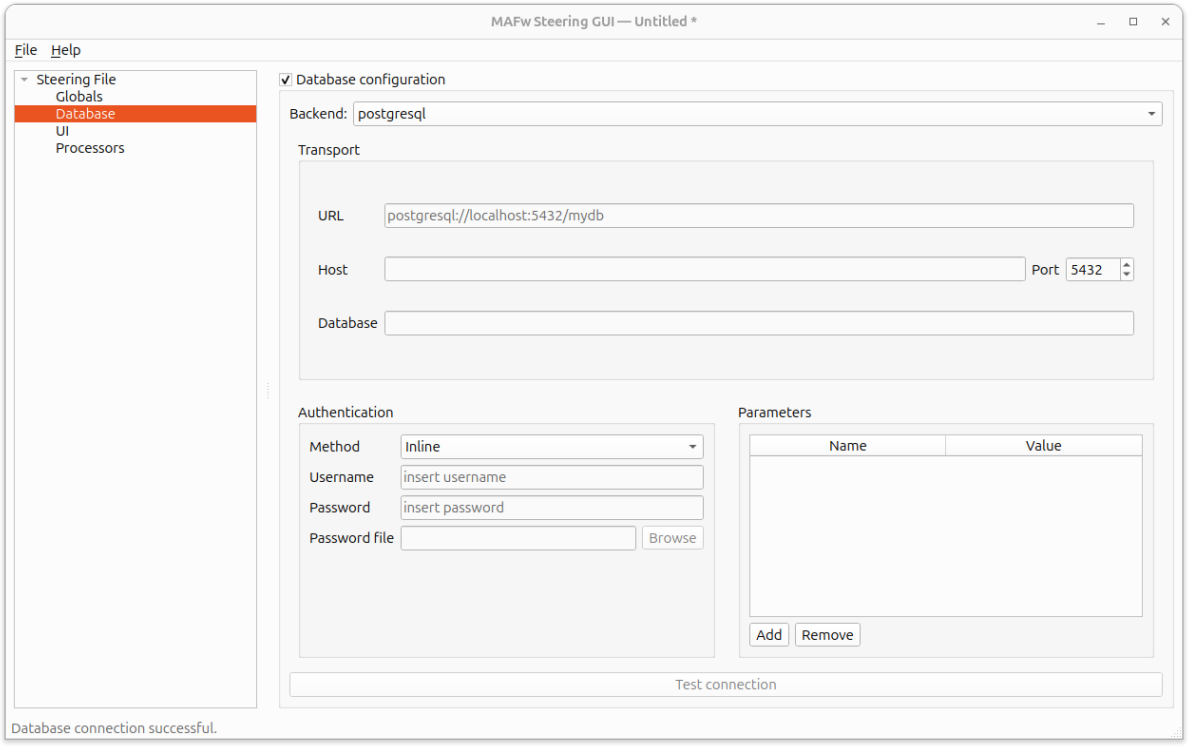
Database Editor

Configure how MAFw connects to your database. You can choose the database backend (SQLite, PostgreSQL, MySQL) and provide the necessary credentials and connection details. Have a look at *here* for more details. Clicking the *Test connection* button you will have the chance to verify if your setup is really able to connect to the database, anticipating possible connection issues.

The database editor is divided into three main widgets resembling the three structural layers in the database section in the steering file: the transport end point (URL), the authentication layer and the parameter section. The URL field in the GUI is read-only, the user should modify either the path (for SQLite) or the host, port and database for the server based databases.

For the authentication, you can choose among the three available methods, **inline** meaning that username and password will be written as plain text in the steering file, **environmental variables**, meaning that instead of encoding the username and password, the steering file will contain the name of environmental variables pointing to the actual username and password. The last, and most secure, option is to provide the path to a password file.

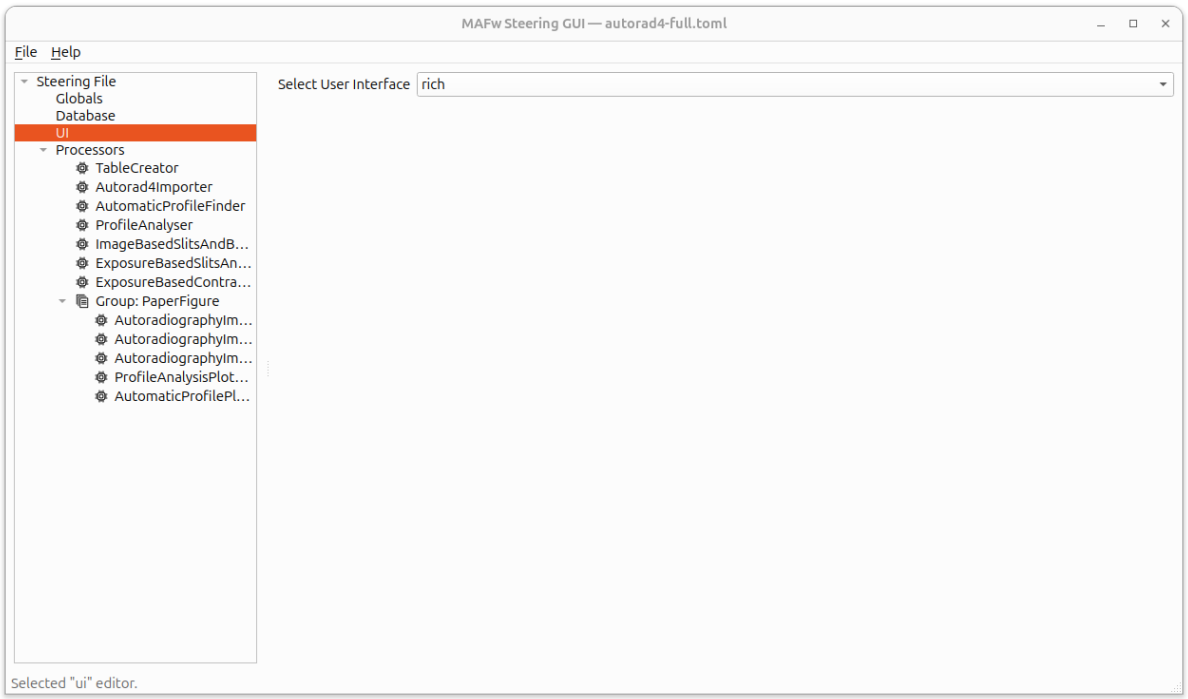
In the parameter section, the user can append other parameters / values to be passed to the peewee connect function. This is something for advanced users, so be aware of what your are doing. In case of doubts, just click on the test connection to see if it works or not.



If you do not want to have any database for your analysis, just unchecked the Database configuration at the very top and no DBConfiguration table will be encoded in the steering file.

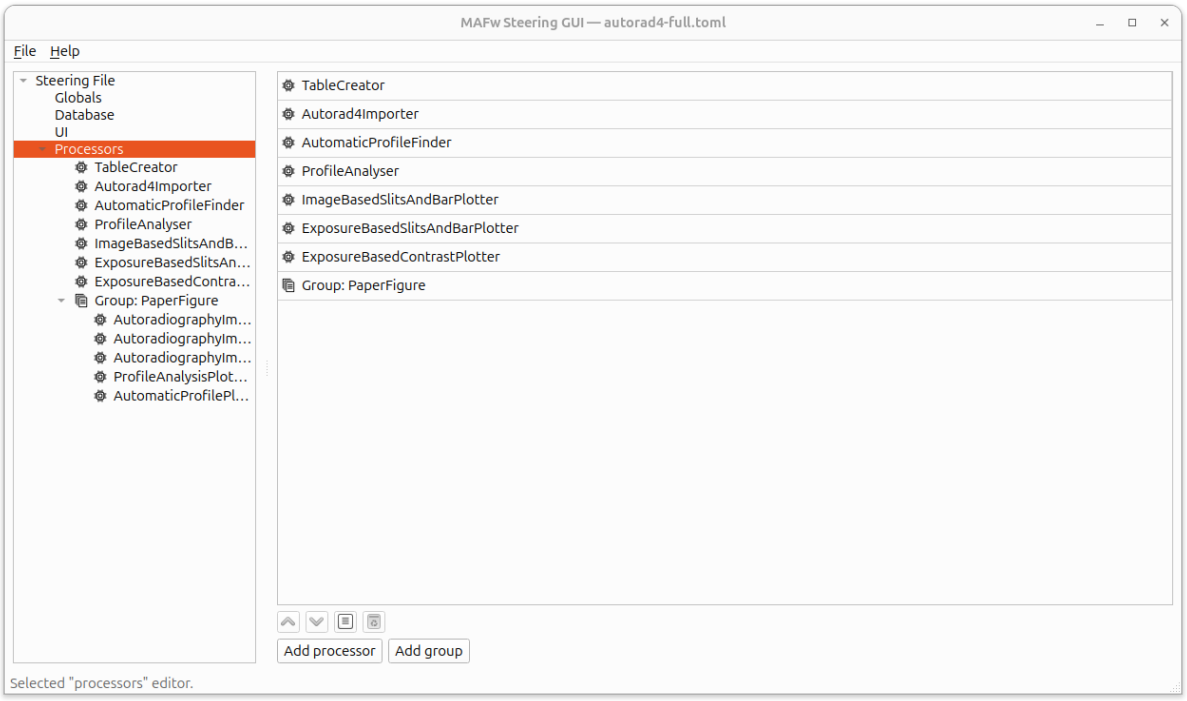
UI Editor

Decide how MAFw should talk to you during execution. You can choose between a simple console output, a rich text interface, or, if you have installed them, other available plugins.



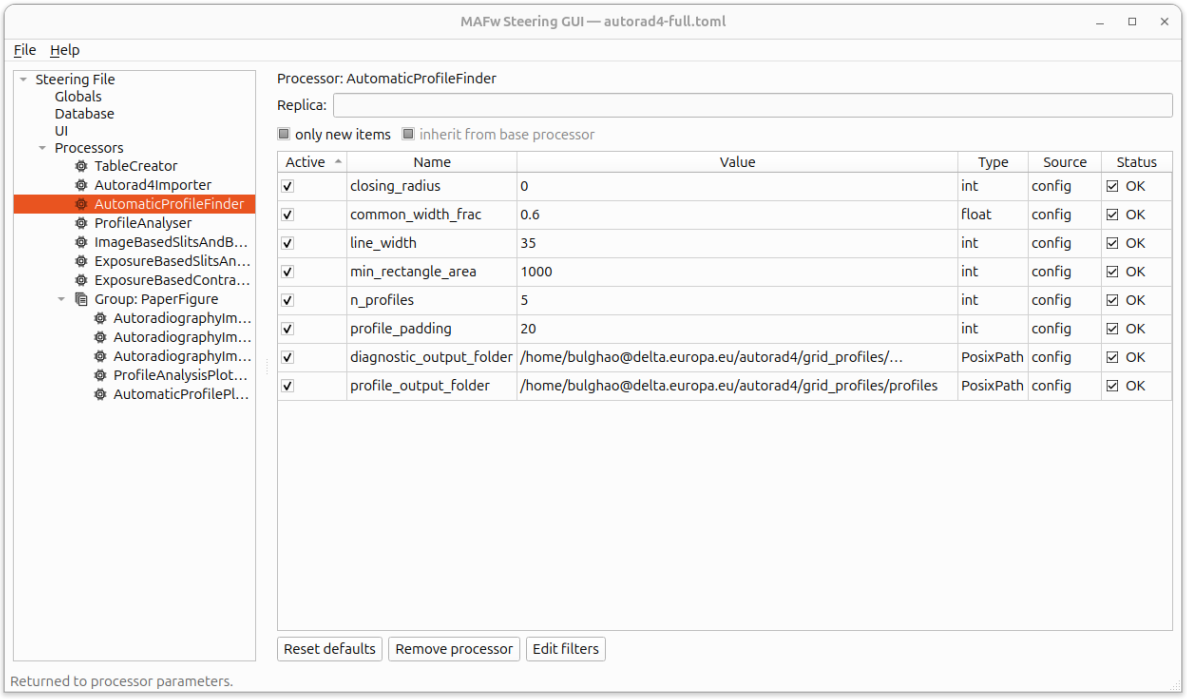
Pipeline Editor

This is where you manage the processors in your analysis. To display this editor you can click on the Processor section of the steering file tree. You can add new processors from the list of available plugins, remove them, or change the order in which they appear in the file. The UI is meant to be self-explanatory and intuitive, so just drag and drop items to reorder them.



Processor Parameter Editor

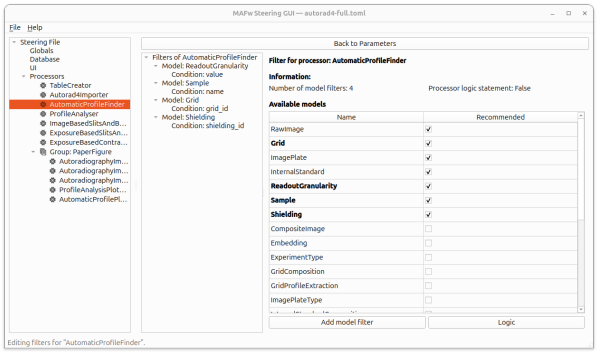
When you select a specific processor, you can assign it a specific replica name and you can edit all its parameters. The GUI dynamically builds an editor based on the processor's requirements, so you'll see checkboxes for booleans, number inputs for integers, and so on. If the parameter name is not clear enough, you can move the mouse over the parameter line and a tool-tip will present you the help text that you have entered in the *processor.ActiveParameter* definition.

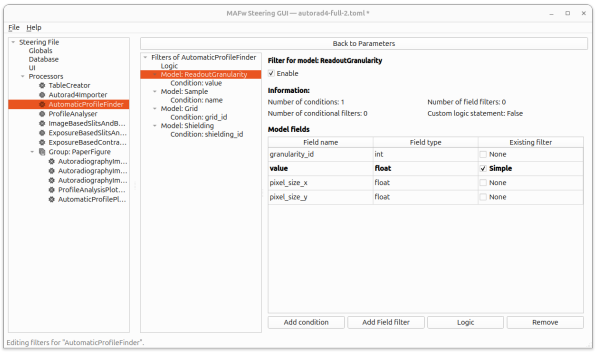


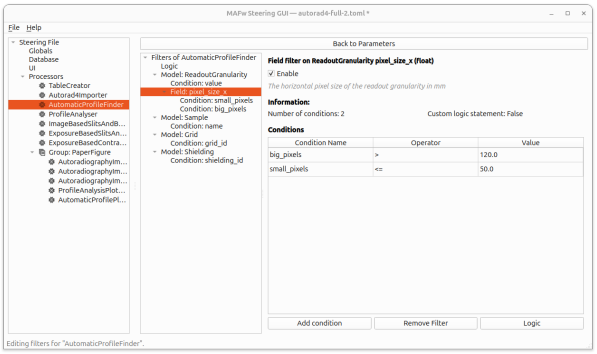
Within the processor settings, you can also manage *filters*:

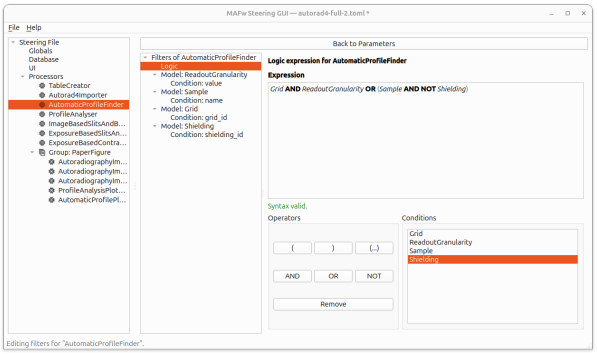
- **Filter Manager**: A central place to see and organize all filters applied to the processor.
- **Model filter editor**: Select which data models this processor should operate on.
- **Field filter editor**: Filter data based on specific field values.
- **Conditional filter editor**: Create more complex rules, like checking if a value is within a range or exists in a predefined list.
- *Logic editor*: The ultimate tool for complex selections. It allows you to combine multiple filters using AND, OR, and NOT logic.

To access the filter manager, just click the Edit filters button for the specific processor and there you can operate directly on the hierarchical structure of MAFw filters. Here you will experience the big advantage of the GUI approach, you do not have to remember the database model names or even worse the field names, the GUI will present you with the available models and available fields. Just follow your instinct and do not forget to explore the context menu. The logic editor is another example of how the GUI can simplify your life. You can either write directly the logical statement and get suggestions for the available names and real time validation, or you can use the buttons and the table to build a sentence just by clicking.



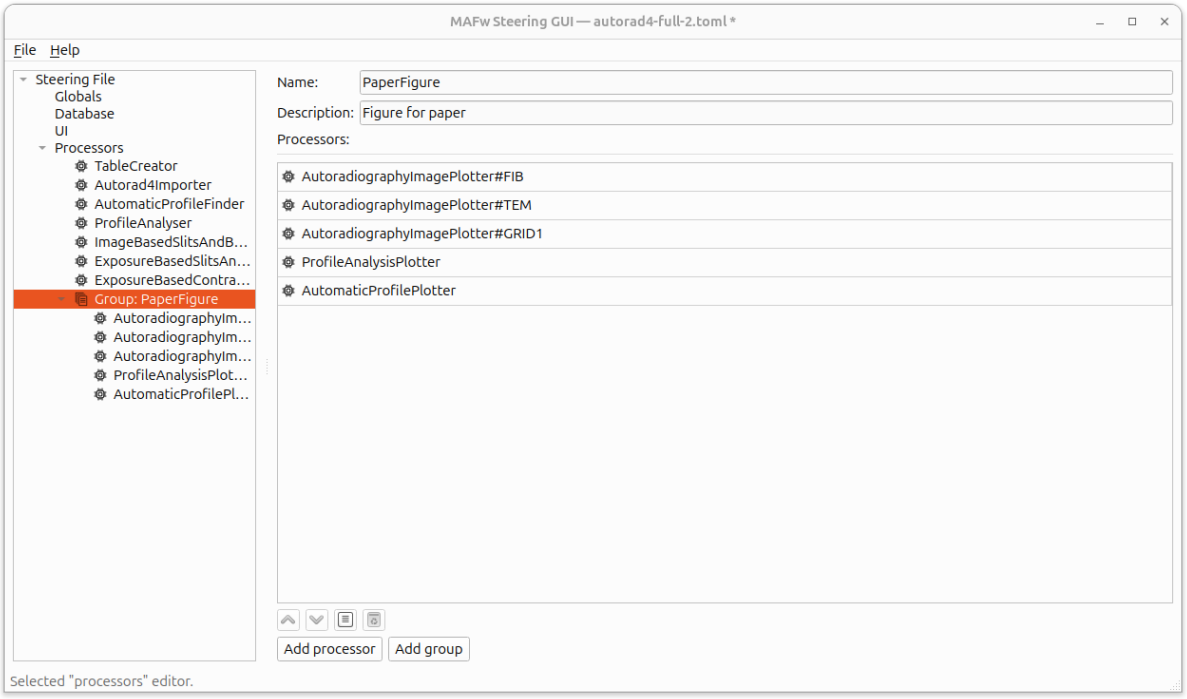






Group Editor

Groups allow you to bundle processors together. This is useful for organizing complex workflows or for running specific parts of your analysis independently. You can pick which processors belong to each group and the GUI will make sure you don't accidentally create any "infinite loops" of groups calling each other.



12.2 The backbone of the GUI: the Steering Builder

> **Note**
>
> This is a very technical chapter that you can skip if you are not interested in developing tools that will programmatically generate steering files.

The Steering GUI is one nice tool that provides an easy and intuitive interface to our in-memory editable steering model, the so called `SteeringBuilder`.

The `mafw.steering` package exposes a TOML-centric builder that is completely decoupled from processor instantiation. It reads and writes steering files while keeping metadata editable via a set of dataclasses, a validation layer, and a *tomlkit* serializer so that any front-end tools can collaborate without touching the execution engine.

For the moment, the builder is mainly used by the Steering GUI, but in the era of Agentic AI, we might provide soon a MCP server exposing the builder API layer so that you would be able to use your favorite AI Agent to build the steering file from a natural language description of what you would like to achieve.

12.2.1 Overview

The MAFw world is divided in two hemispheres: we have the *so-called* execution framework, where the **Runner** is reading a steering file in TOML format, parsing a configuration dictionary out of it, build up and configure the pipeline accordingly and finally run it!

The other hemisphere is where the steering file is generated, and here the main role is played by the **Builder**. This is a in-memory editable model of the steering file and the user can build and edit each and every element of the steering file interacting with the API. When finished, the builder content can be serialized generating a validated steering file that can be provided to the Runner.

In other words, the TOML steering file is the only point of contact of the two hemispheres. hand, or by an MCP agent.

12.2.2 Builder API

- `SteeringBuilder` tracks globals, per-processor sections, groups, database settings, and interface configuration without ever touching a processor class.
- Processor helpers can add/remove references, manage replicas, tune parameters, and edit filters (fields, logic strings, and conditionals).
- Group helpers keep `processors_to_run` lists readable and detect cyclic definitions early.
- The builder exposes `to_document`/`write` to hand TOML documents to other code or to disk.

12.2.3 Serialization & Validation

- `serializer.serialize` emits TOML via *tomlkit*, preserving insertion order and honoring the required section order: global scalars, `DBConfiguration`, processor sections, replicas, group sections, and finally `UserInterface`.
- Filters appear as `[Processor.__filter__.Model]` sections. Replica tables optionally emit `__inheritance__ = false` when requested.
- `validation.validate` enforces structural rules such as non-empty `processors_to_run`, valid references, and absence of group cycles. No processor metadata is instantiated or expanded.

CHAPTER 13

DEVELOPMENT TOOLS

MAFw ships with a unified **development tool** called `devtools` that supports maintenance tasks such as building versioned documentation trees, preparing tagged releases, and verifying dependency compatibility.

These tools are **not** part of the scientific user-facing workflow: a typical MAFw user does not need them, and can safely ignore this section. They exist for the benefit of the development team and for community members who want to contribute to MAFw in a consistent, reproducible way.

13.1 Requirements

The `devtools` CLI requires additional Python packages that are installed when MAFw is set up with the optional `[devtools]` or `[dev]` feature:

```
pip install mafw[devtools]
```

Dependency	Minimum supported version	Description
packaging	>=26.2, <27	to parse and compare dependency version specifiers
requests	>=2.34.0, <3	to interact with the GitLab API for registry operations
psutil	>=7.2.2, <8	to manage background processes (documentation server)
ruamel.yaml	>=0.18.14, <0.19	to read and write YAML files preserving comments and formatting (pre-commit config)

13.2 How to invoke `devtools`

The unified tool is available as a console entry point from an activated development environment:

```
devtools --help
```

Or via Hatch (recommended for CI/CD and for reproducible local runs):

```
hatch run dev.py3.14:devtools --help
```

> **Note**
>
> The Hatch environment name includes the Python version (for example `dev.py3.14`). On CI, the version is typically driven by the `PYTHON_VERSION` variable in `.gitlab-ci.yml`.

13.3 Command structure

```
devtools
├── completion       Manage shell completion
│   ├── install
│   ├── uninstall
│   └── show
├── documentation    Build and manage versioned documentation
│   ├── build          Build multiversion documentation tree
│   ├── current        Build only the current working tree
│   ├── clean          Remove output directories
│   ├── prune          Remove old versions to respect a size limit
│   ├── redirects      Generate GitLab Pages _redirects file
│   ├── landing        Generate root landing page
│   ├── requirements   Generate RST requirement files from pyproject.toml
│   ├── registry       Interact with the GitLab Generic Package Registry
│   │   ├── upload
│   │   ├── download
│   │   └── delete
│   └── server         Local HTTP server helper
│       ├── start
│       ├── status
│       ├── stop
│       └── restart
├── release          Release lifecycle management
│   └── create        Prepare a new tagged release
├── dependencies     Dependency verification and maintenance
│   ├── freeze         Freeze dependency upper bounds in pyproject.toml
│   ├── unfreeze       Remove frozen dependency upper bounds
│   ├── latest         Rolling compatibility for the latest dependencies
│   │   ├── check        Verify compatibility with newest dependencies
│   │   └── compare      Compare dependencies against reference pylock files
│   ├── oldest         Oldest dependency stack verification
│   │   └── check        Verify compatibility with oldest supported dependencies
│   ├── audit          Audit dependency vulnerabilities using pip-audit
│   └── registry       GitLab registry for dependency reference files
│       ├── upload
│       ├── download
│       ├── delete
│       └── prune
└── toolchain         Centralized development tool version management
    ├── list             Show all managed tools
    ├── check            Compare current vs latest versions
    ├── update           Bring tools to their latest versions
    ├── verify           Check configuration consistency
    └── bootstrap        Install missing host tools
```

13.4 Shell completion

`devtools` exposes shell TAB completion through the `completion` command group. The feature is available for `bash`, `zsh` and `fish`.

Install it in the active virtual environment with:

```
devtools completion install
```

If you only want to activate directly (but not permanently) the completion functionality, use:

```
eval "$(devtools completion show)"
```

13.5 Documentation commands

13.5.1 Purpose and idea

The `documentation` group builds the MAFw documentation using Sphinx and produces a directory tree that contains:

- One folder per stable tag (for example `v2.1.0`).
- A `stable` alias pointing to the latest stable tag.
- Optionally a `dev` alias for the current branch if it is ahead of `stable`.
- A `latest` folder containing a build of the current working tree.

This makes it possible to publish *multiple* documentation versions (HTML and optionally PDF) in a single GitLab Pages site, while keeping the workflow reproducible and CI-friendly.

13.5.2 Typical usage examples

Build only the current documentation (`current`)

This is the fastest way to validate documentation changes on your current branch.

```
devtools documentation current -y
```

Rebuild from scratch:

```
devtools documentation current -y --from-scratch
```

Build a multiversion documentation tree (`build`)

```
devtools documentation build --min-vers v1.0.0 --build-pdf
```

How documentation is built on CI

In the GitLab pipeline, the multiversion documentation is generated by the `doc_build_all` job:

```
hatch run dev.py${PYTHON_VERSION}:devtools documentation build \
  --min-vers $DOC_MIN_VERS \
  --build-pdf \
  --max-size $DOC_MAX_SIZE \
  --zip-filepath /tmp/zips \
  --with-zip-file \
  --with-upload-zip \
  --with-cached-packages
```

Start a local documentation server

After building the documentation locally, serve the generated tree over HTTP:

```
devtools documentation server start -d docs/build -p 8000
```

Even though one can browse directly the generated files in the browser (*file:///*), serving the pages via a real server assures the proper functioning of all JavaScript and version switching mechanism.

13.6 Release commands

13.6.1 Purpose and idea

The `release` group orchestrates the MAFw release lifecycle: version bumps, changelog updates, and tagging. It enforces consistency and automates repetitive tasks.

13.6.2 Release reproducibility and rolling compatibility

MAFw uses open lower bounds during development (for example `rich>=13.9.4`) to allow working against newer upstream releases. For tagged releases, however, reproducibility matters.

The `release create` command applies a **freeze** → **release** → **unfreeze** pattern:

1. **Freeze**: rewrite dependency specifiers with explicit upper bounds based on the highest resolved version.
2. **Release**: the release commit includes the frozen `pyproject.toml`.
3. **Unfreeze**: remove the computed upper bounds so development continues unrestricted.

13.6.3 Documentation target version

The documentation target version is the `major.minor` value used by the release workflow to annotate public API docstrings with Sphinx directives such as `versionadded` and `versionchanged`.

13.6.4 Prepare a new release (`release create`)

The `release create` command automates the entire release pipeline. It requires a positional argument `SEGMENTS` which is passed to Hatch (e.g., `minor,rc`, `rc`, or `release`).

```
devtools release create minor,rc --dry-run
```

Logical workflow summary

1. **Safety checks**: Verify the current branch is `main` and the working tree is clean.
2. **Freeze**: Rewrite `pyproject.toml` to add explicit upper bounds to dependencies.
3. **Bump**: Update the project version using `hatch version`.
4. **Doc target**: Update the documentation target version used by docstring version directives.
5. **Metadata**: Update `NOTICE.txt` and `CHANGELOG.md`.
6. **Requirements**: Update the requirements RST files and `README.rst` with frozen dependencies.
7. **Notes**: Optionally generate a `release_note_vX.Y.Z.md` file.
8. **Tag**: Commit the changes and create a local git tag.
9. **Unfreeze**: Remove the computed upper bounds from `pyproject.toml`.
10. **Requirements**: Restore the requirements RST files and `README.rst` to open upper bounds.
11. **Commit**: Create a second commit on `main` with the unfrozen dependencies.
12. **Push**: Optionally push both commits and the tag to the remote repository.

13.7 Dependencies commands

13.7.1 Purpose and idea

The `dependencies` group ensures MAFw is compatible with both the latest and oldest supported dependency stacks. It also provides tools for freezing upper bounds, auditing vulnerabilities, and managing reference files in the GitLab Generic Package Registry.

13.7.2 Verify dependency stacks

Verify against newest dependencies:

```
devtools dependencies latest check
```

Verify against oldest supported dependencies (using `lowest-direct` resolution):

```
devtools dependencies oldest check
```

Both commands support `--preserve-envs` to keep hatch environments for local debugging.

Deprecated since version 2.3: The `--remove-envs/--no-remove-envs` option on `dependencies oldest check` is deprecated. Use `--preserve-envs` instead.

13.7.3 Compare dependency versions (`dependencies latest compare`)

The `compare` subcommand detects ecosystem drift by comparing the latest resolved dependency versions against reference pylock files.

```
devtools dependencies latest compare
devtools dependencies latest compare -o drift-report.md -f markdown
devtools dependencies latest compare -o drift-report.json -f json
devtools dependencies latest compare --gitlab-ref
```

13.7.4 Freeze and unfreeze

```
devtools dependencies freeze
devtools dependencies unfreeze
```

13.7.5 Manage dependency reference files (`dependencies registry`)

To speed up CI/CD pipelines, `devtools` can cache dependency reference files in the GitLab Generic Package Registry.

```
devtools dependencies registry upload --all
devtools dependencies registry download --all
devtools dependencies registry prune
```

Added in version 2.3: The `dependencies registry prune` command.

13.7.6 Vulnerabilities check (`dependencies audit`)

A scheduled pipeline executed every night verifies that there are no known vulnerabilities for all MAFw dependencies. To run the audit manually:

```
devtools dependencies audit
```

13.8 Toolchain commands

13.8.1 Purpose and idea

The `toolchain` group centralizes management of all development tool versions used by MAFw. It covers four main activities:

- **Checking** whether tools are at their latest versions.
- **Updating** tools following their declared update policy (pipx upgrade for host tools, lower-bound bump for project tools).

- **Verifying** configuration consistency across `pyproject.toml` and `.pre-commit-config.yaml`.
- **Bootstrapping** missing tools for a fresh development environment.

Tools are categorized as either *host* (installed system-wide via pipx) or *project* (managed through `pyproject.toml`). By default, only project tools are processed; host tools are included explicitly via `--include-host`.

13.8.2 Command tree

```
devtools toolchain
├── list       - Show all managed tools
├── check      - Compare current vs latest versions
├── update     - Bring tools to their latest versions
├── verify     - Check configuration consistency
└── bootstrap  - Install missing host tools
```

13.8.3 Group-level options

The `toolchain` group accepts two mutually exclusive flags that control whether host tools (installed via pipx) participate in subcommand processing:

`--include-host`
: Include host tools (hatch, uv) in the operation. When specified, both host and project tools are processed.

`--exclude-host`
: Explicitly exclude host tools from the operation. This is the default behavior when neither option is provided.

If both `--include-host` and `--exclude-host` are specified, the CLI rejects the invocation with an error message.

13.8.4 Subcommand-level filtering options

Each subcommand (`list`, `check`, `update`, `verify`, `bootstrap`) accepts two repeatable options for filtering individual tools by name:

`-i` / `--include <name>`
: Process **only** the named tools. Can be repeated to include multiple tools. Unrecognized names produce a warning but do not abort.

`-e` / `--exclude <name>`
: Process **all applicable tools except** the named ones. Can be repeated to exclude multiple tools.

These two options are mutually exclusive: specifying both `-i` and `-e` in the same invocation results in an error.

Example — update only ruff and mypy:

```
devtools toolchain update -i ruff -i mypy
```

Example — check all project tools except sphinx:

```
devtools toolchain check -e sphinx
```

13.8.5 List (`toolchain list`)

Displays a table of all applicable managed tools showing their name and category.

```
devtools toolchain list
```

To include host tools in the listing:

```
devtools toolchain --include-host list
```

13.8.6 Check (`toolchain check`)

Compares the currently configured version of each tool against its latest available version and reports whether tools are in sync. Exits with a non-zero exit code if any tool is out of date or if a version detection error occurred.

```
devtools toolchain check
```

Include host tools in the check:

```
devtools toolchain --include-host check
```

13.8.7 Update (`toolchain update`)

Brings tools to their latest versions following each tool's update policy. After a successful update, a post-update validation hook runs (for example, the full test suite for pytest, or `ruff check` for ruff). If validation fails, the update is reverted automatically.

```
devtools toolchain update
```

Update only a specific tool:

```
devtools toolchain update -i ruff
```

Include host tools in the update:

```
devtools toolchain --include-host update
```

13.8.8 Verify (`toolchain verify`)

Checks that all tool configurations are internally consistent. For example, it detects when the ruff version in `pyproject.toml` differs from the `rev` field in `.pre-commit-config.yaml`, or when the pipx-installed uv version does not satisfy the declared constraint in the hatch-uv environment. Exits with code 1 if any inconsistency is found.

```
devtools toolchain verify
```

13.8.9 Bootstrap (`toolchain bootstrap`)

Installs missing tools for a fresh development environment. For host tools, this runs `pipx install`; project tools that are already declared in `pyproject.toml` are typically available after a `pip install -e .[dev]` and are skipped. Exits with a non-zero code if any installation fails.

```
devtools toolchain --include-host bootstrap
```

13.8.10 Managed tools

The following table lists all tools managed by the toolchain command group:

Table 13.1: Managed tools

Tool	Category	Update policy
hatch	host	`pipx upgrade`
uv	host	`pipx upgrade` + pyproject.toml hatch-uv env sync
pytest	project	Lower bound bump in `test` optional-dependencies
mypy	project	Lower bound bump in `types` env extra-dependencies
sphinx	project	Lower bound bump in `doc` optional-dependencies
ruff	project	Lower bound bump in `dev` optional-dependencies + pre-commit rev sync
pre-commit	project	Lower bound bump in `dev` optional-dependencies
git-cliff	project	Lower bound bump in `dev` optional-dependencies
pip-audit	project	Lower bound bump in `dev` optional-dependencies

CHAPTER 14

API

This page is particularly useful for the developers because it offers a quick view at the implemented API.

The version directives embedded in docstrings provide an indicative history of API evolution only. For the most precise chronology, consult the release notes and the git history, which can capture sub-release granularity that docstrings intentionally do not.

mafw	The Modular Analysis Framework

14.1 mafw

The Modular Analysis Framework

A software tool for scientists written by scientists!

Module Attributes

mafw_hookimpl	Marker to be imported and used in plugins loading

mafw.**mafw_hookimpl = <pluggy._hooks.HookimplMarker object>**

Marker to be imported and used in plugins loading

Modules

active	Module implements active variable for classes.
db	Defines the DB interface
decorators	The module provides some general decorator utilities that are used in several parts of the code, and that can be reused by the user community.
devtools	MAFw development tools package.
enumerators	Module provides a set of enumerators for dealing with standard tasks.
examples	A library of examples.
hookspecs	Defines the hook specification decorator bound the MAFw library.

continues on next page

Table 14.3 – continued from previous page

lazy_import	This module provides classes for lazy importing of plugins, ensuring thread-safe access and transparent usage.
mafw_errors	Module defines MAFw exceptions
models	Model helpers shared across MAFw.
plugin_manager	Plugin management system for MAFw framework.
plugins	Exports Processor classes to the execution script.
processor	Processor package — core building block of the MAFw pipeline.
processor_library	Library of generic processors.
runner	Provides a container to run configurable and modular analytical tasks.
scripts	Executables.
steering	Steering builder tools for MAFw.
steering_gui	Application layer for steering GUI orchestration.
timer	Module implements a simple timer to measure the execution duration.
tools	The package provides a set of tools for different range of applications.
ui	User interface modules.

14.1.1 mafw.active

Module implements active variable for classes.

Module Attributes

ActiveType	A type for templating the Active class.
ActivableType	A type for all classes that can include an Active.

Classes

Active([default])	A descriptor class to make class variable **active**.

class mafw.active.**ActivableType**

A type for all classes that can include an Active.

alias of TypeVar('ActivableType')

class mafw.active.**Active**(*default:* ActiveType | *None* = *None*)

Bases: Generic[*ActiveType*]

A descriptor class to make class variable **active**.

When assigned to a class variable, any change of this value will trigger a call back to a specific function.

Here is a clarifying example.

```
class Person:
    age = Active()

    def __init__(self, age):
        self.age = age

    def on_age_change(self, old_value, new_value):
        # callback invoked every time the value of age is changed
```

(continues on next page)

(continued from previous page)

```
        # do something with the old and the new age
        pass

    def on_age_set(self, value):
        # callback invoked every time the value on age is set to the same
        # value as before.
        pass

    def on_age_get(self, value):
        # callback invoked every time the value of age is asked.
        # not really useful, but...
        pass
```

Once you have assigned an Active to a class member, you need to implement the callback in your class. If you do not implement them, the code will run without problems.

The three callbacks have the signature described in the example.

- on_[var_name]_change(self, old, new)
- on_[var_name]_set(self, value)
- on_[var_name]_get(self, value)

Constructor parameter:

Parameters
default (ActiveType) – Initial value of the Active value. Defaults to None.

class mafw.active.**ActiveType**

A type for templating the Active class.

alias of TypeVar('ActiveType')

14.1.2 mafw.db

Defines the DB interface

Modules

db_configurations	Module provides default configurations for different database engines.
db_connection	Database connection configuration helpers.
db_filter	Database filter module for MAFW.
db_model	The module provides functionality to MAFw to interface to a DB.
db_types	Database Type Definitions
db_wizard	The module allows to generate a DB model structure starting from an existing DB.
fields	Module provides customised model fields specific for MAFw.
model_register	Module for registering and retrieving database models.
std_tables	Module provides standard tables that are included in all database created by MAFw processor.
trigger	Module provides a Trigger class and related tools to create triggers in the database via the ORM.

mafw.db.db_configurations

Module provides default configurations for different database engines.

Module Attributes

default_conf	default configuration dictionary used to generate steering files
db_scheme	default database scheme
DEFAULT_SQLITE_PRAGMAS	Default pragma settings for SQLite connections.

mafw.db.db_configurations.**DEFAULT_SQLITE_PRAGMAS = {'cache_size': -64000, 'foreign_keys': 1, 'journal_mode': 'wal', 'synchronous': 0}**

Default pragma settings for SQLite connections.

mafw.db.db_configurations.**db_scheme = {'mysql': 'mysql://', 'postgresql': 'postgresql://', 'sqlite': 'sqlite:///'}**

default database scheme

mafw.db.db_configurations.**default_conf: dict[str, dict[str, Any]] = {'mysql': {'URL': 'mysql://user:passwd@ip:port/my_db', 'authentication': {'method': 'env', 'password': 'MYSQL_PASS', 'username': 'MYSQL_USER'}, 'parameters': {'mysql': {}}}, 'postgresql': {'URL': 'postgresql://postgres:my_password@localhost:5432/my_database', 'authentication': {'method': 'env', 'password': 'POSTGRES_PASS', 'username': 'POSTGRES_USER'}, 'parameters': {'postgresql': {}}}, 'sqlite': {'URL': 'sqlite:///my_database.db', 'parameters': {'sqlite': {'pragmas': {'cache_size': -64000, 'foreign_keys': 1, 'journal_mode': 'wal', 'synchronous': 0}}}}}**

default configuration dictionary used to generate steering files

mafw.db.db_connection

Database connection configuration helpers.

Author
Bulgheroni Antonio

Description
Normalize database steering configuration into peewee connection arguments.

Module Attributes

VALID_AUTH_METHODS	Supported authentication methods for `DBConfiguration.authentication`.
LEGACY_KEYS	Keys reserved by the legacy DBConfiguration schema.

Functions

build_connection_parameters(config)	Normalize DBConfiguration mappings into peewee connection arguments.

mafw.db.db_connection.**build_connection_parameters**(*config: Mapping[str, Any]*) → tuple[str, dict[str, Any]]

Normalize DBConfiguration mappings into peewee connection arguments.

The input can either be the full steering configuration dictionary or the nested `DBConfiguration` table. The returned tuple provides the URL and the keyword arguments for `peewee.connect`.

Parameters
config (*Mapping[str, Any]*) – Steering configuration or DBConfiguration table.

Returns
Tuple with database URL and connection keyword arguments.

Return type
tuple[str, dict[str, Any]]

Raises
ValueError – When required fields are missing or invalid.

`mafw.db.db_connection.LEGACY_KEYS: set[str] = {'URL', 'pragmas'}`
Keys reserved by the legacy DBConfiguration schema.

`mafw.db.db_connection.VALID_AUTH_METHODS: set[str] = {'env', 'file', 'inline'}`
Supported authentication methods for `DBConfiguration.authentication`.

mafw.db.db_filter

Database filter module for MAFW.

This module provides classes and utilities for creating and managing database filters using Peewee ORM. It supports various filtering operations including simple conditions, logical combinations, and conditional filters where one field's criteria depend on another.

The module implements a flexible filter system that can handle:

- Simple field comparisons (equality, inequality, greater/less than, etc.)
- Complex logical operations (AND, OR, NOT)
- Conditional filters with dependent criteria
- Nested logical expressions
- Support for various data types and operations

Key components include:

- *FilterNode*: Abstract base class for filter nodes
- *ConditionNode*: Represents individual field conditions
- *LogicalNode*: Combines filter nodes with logical operators
- *ConditionalNode*: Wraps conditional filter conditions
- *ModelFilter*: Main class for building and applying filters to models
- *ProcessorFilter*: Container for multiple model filters in a processor

The module uses a hierarchical approach to build filter expressions that can be converted into Peewee expressions for database queries. It supports both simple and complex filtering scenarios through a combination of direct field conditions and logical expressions.

Changed in version v2.0.0: Major overhaul introducing conditional filters and logical expression support.

Example usage:

```
from mafw.db.db_filter import ModelFilter

# Create a simple filter
flt = ModelFilter(
    'Processor.__filter__.Model',
    field1='value1',
    field2={'op': 'IN', 'value': [1, 2, 3]},
)
```

(continues on next page)

(continued from previous page)

```
# Bind to a model and generate query
flt.bind(MyModel)
query = MyModel.select().where(flt.filter())
```

See also

peewee - The underlying ORM library used for database operations

LogicalOp - Logical operation enumerations used in filters

Module Attributes

Token	Type definition for a logical expression token
NameNode	An atom is a tuple of the literal string 'NAME' and the value
NotNode	A NOT node is a tuple of 'NOT' and a recursive node
BinaryNode	AND/OR nodes are tuples of the operator and two recursive nodes
ExprNode	The main recursive type combining all options
TOKEN_SPECIFICATION	Token specifications
MASTER_RE	Compiled regular expression to interpret the logical expression grammar

Functions

ast_to_string(ast)	Convert an abstract syntax tree (AST) back to its string representation.
tokenize(text)	Tokenize a logical expression string into a list of tokens.

Classes

ConditionNode(field, operation, value[, name])	Represents a single condition node in a filter expression.
ConditionalFilterCondition(condition_field, ...)	Represents a conditional filter where one field's criteria depends on another.
ConditionalNode(conditional[, name])	Wraps *ConditionalFilterCondition* behaviour as a *FilterNode*.
ExprParser(text, *[, valid_names])	Recursive descent parser producing a simple Abstract Syntax Tree (AST).
FilterNode()	Abstract base for nodes.
LogicalNode(op, *children)	Logical combination of child nodes.
ModelFilter(name_, **kwargs)	Class to filter rows from a model.
ProcessorFilter([data])	A special dictionary to store all *Filters* in a processors.

Exceptions

MissingTokenError(expected, *[, position])	Raised when a specific token was required but missing.

continues on next page

Table 14.13 – continued from previous page

ParseError(message, *[, position])	Base exception for logical expression parsing failures.
UnexpectedEndOfExpressionError(*[, ...])	Raised when the expression ends before the parser could finish.
UnexpectedTokenError(token, *[, expected, ...])	Raised when a token is present but not valid in the current context.
UnknownNameError(name, *, valid_names[, ...])	Raised when a NAME token is not in the supplied whitelist.

exception mafw.db.db_filter.**MissingTokenError**(*expected: str | Sequence[str]*, ** (Keyword-only parameters separator (PEP 3102))*, *position: int | None = None*)

Bases: *ParseError*

Raised when a specific token was required but missing.

exception mafw.db.db_filter.**ParseError**(*message: str*, ***, *position: int | None = None*)

Bases: ValueError

Base exception for logical expression parsing failures.

exception mafw.db.db_filter.**UnexpectedEndOfExpressionError**(***, *expected: str | Sequence[str] | None = None*, *position: int | None = None*)

Bases: *ParseError*

Raised when the expression ends before the parser could finish.

exception mafw.db.db_filter.**UnexpectedTokenError**(*token: tuple[str, str]*, ***, *expected: str | Sequence[str] | None = None*, *position: int | None = None*)

Bases: *ParseError*

Raised when a token is present but not valid in the current context.

exception mafw.db.db_filter.**UnknownNameError**(*name: str*, ***, *valid_names: Iterable[str]*, *position: int | None = None*)

Bases: *ParseError*

Raised when a NAME token is not in the supplied whitelist.

class mafw.db.db_filter.**ConditionNode**(*field: str | None*, *operation:* LogicalOp *| str*, *value: Any*, *name: str | None = None*)

Bases: *FilterNode*

Represents a single condition node in a filter expression.

This class encapsulates a single filtering condition that can be applied to a model field. It supports various logical operations through the *LogicalOp* enumerator or string representations of operations.

Added in version v2.0.0.

Initialize a condition node.

Parameters

- **field** (*str | None*) – The name of the field to apply the condition to.
- **operation** (*LogicalOp | str*) – The logical operation to perform.
- **value** (*Any*) – The value to compare against.
- **name** (*str | None, Optional*) – Optional name for this condition node.

to_expression(*model: type[Model]*) → Expression

Convert this condition node to a Peewee expression.

This method translates the condition represented by this node into a Peewee expression that can be used in database queries.

Parameters
model (*type[Model]*) – The model class containing the field to filter.

Returns
A Peewee expression representing this condition.

Return type
peewee.Expression

Raises

- **RuntimeError** – If the node has no field to evaluate.
- **ValueError** – If an unsupported operation is specified.
- **TypeError** – If operation requirements are not met (e.g., IN operation requires list/tuple).

class mafw.db.db_filter.**ConditionalFilterCondition**(*condition_field: str*, *condition_op: str | LogicalOp*, *condition_value: Any*, *then_field: str*, *then_op: str | LogicalOp*, *then_value: Any*, *else_field: str | None = None*, *else_op: str | LogicalOp | None = None*, *else_value: Any | None = None*, *name: str | None = None*)

Bases: `object`

Represents a conditional filter where one field's criteria depends on another.

This allows expressing logic like: "IF field_a IN [x, y] THEN field_b IN [1, 2] ELSE no constraint on field_b"

Example usage:

```
# Filter: sample_id in [1,2] if composite_image_id in [100,101]
condition = ConditionalFilterCondition(
    condition_field='composite_image_id',
    condition_op='IN',
    condition_value=[100, 101],
    then_field='sample_id',
    then_op='IN',
    then_value=[1, 2],
)

# This generates:
# WHERE (composite_image_id IN (100, 101) AND sample_id IN (1, 2))
#    OR (composite_image_id NOT IN (100, 101))
```

Initialise a conditional filter condition.

Parameters

- **condition_field** (*str*) – The field to check for the condition
- **condition_op** (*str | LogicalOp*) – The operation for the condition (e.g., 'IN', '==')
- **condition_value** (*Any*) – The value(s) for the condition
- **then_field** (*str*) – The field to filter when condition is true
- **then_op** (*str | LogicalOp*) – The operation to apply when condition is true

- **then_value** (*Any*) – The value(s) for the then clause
- **else_field** (*str | None*) – Optional field to filter when condition is false
- **else_op** (*str | LogicalOp | None*) – Optional operation when condition is false
- **else_value** (*Any | None*) – Optional value(s) for the else clause
- **name** (*str | None, Optional*) – The name of this condition. Avoid name clashing with model fields. Defaults to None

to_expression(*model: type[Model]*) → Expression

Convert this conditional filter to a Peewee expression.

The resulting expression is: (condition AND then_constraint) OR (NOT condition AND else_constraint)

Which logically means:

- When condition is true, apply then_constraint
- When condition is false, apply else_constraint (or no constraint)

Parameters
model (*type[Model]*) – The model class containing the fields

Returns
A Peewee expression

Return type
peewee.Expression

class mafw.db.db_filter.**ConditionalNode**(*conditional: ConditionalFilterCondition*, *name: str | None = None*)

Bases: *FilterNode*

Wraps *ConditionalFilterCondition* behaviour as a *FilterNode*.

This class serves as an adapter to integrate conditional filter conditions into the filter node hierarchy, allowing them to be treated uniformly with other filter nodes during expression evaluation.

Added in version v2.0.0.

Initialize a conditional node.

Parameters

- **conditional** (ConditionalFilterCondition) – The conditional filter condition to wrap
- **name** (*str | None, Optional*) – Optional name for this conditional node

to_expression(*model: type[Model]*) → Expression

Convert this conditional node to a Peewee expression.

This method delegates the conversion to the wrapped conditional filter condition's *to_expression()* method.

Parameters
model (*type[Model]*) – The model class to generate the expression for

Returns
A Peewee expression representing this conditional node

Return type
peewee.Expression

class mafw.db.db_filter.**ExprParser**(*text: str*, *, *valid_names: Iterable[str] | None = None*)

Bases: object

Recursive descent parser producing a simple Abstract Syntax Tree (AST).

The parser handles logical expressions with the following grammar:

```
expr    := or_expr
or_expr := and_expr ("OR" and_expr)*
and_expr:= not_expr ("AND" not_expr)*
not_expr:= "NOT" not_expr | atom
atom    := NAME | "(" expr ")"
```

AST nodes are tuples representing different constructs:

- ("NAME", "token"): A named element (field name or filter name)
- ("NOT", node): A negation operation
- ("AND", left, right): An AND operation between two nodes
- ("OR", left, right): An OR operation between two nodes

Added in version v2.0.0.

To help users diagnose grammar and semantic problems, the parser now reports detailed error classes with character offsets and accepts an optional `valid_names` iterable to reject unknown identifiers early.

Initialize the expression parser with a logical expression string.

Parameters

- **text** (*str*) – The logical expression to parse
- **valid_names** (*Iterable[str] | None*) – Optional whitelist of valid NAME tokens

_current_position() → int

Return the character offset of the next token or the end of input.

accept(**kinds: str*) → tuple[str, str] | None

Accept and consume the next token if it matches one of the given types.

Parameters
kinds (*str*) – Token types to accept

Returns
The consumed token if matched, otherwise None

Return type
Token | None

expect(*kind: str*) → tuple[str, str]

Expect and consume a specific token type.

Parameters
kind (*str*) – The expected token type

Returns
The consumed token

Return type
Token

Raises
ParseError – If the expected token is not found

parse() → tuple[Literal['NAME'], str] | tuple[Literal['NOT'], ExprNode] | tuple[Literal['AND', 'OR'], ExprNode, ExprNode]

Parse the entire logical expression and return the resulting AST.

Returns
The abstract syntax tree representation of the expression

Return type
ExprNode

Raises
ParseError – If the expression is malformed

parse_and() → tuple[Literal['NAME'], str] | tuple[Literal['NOT'], ExprNode] | tuple[Literal['AND', 'OR'], ExprNode, ExprNode]

Parse an AND expression.

Returns
The parsed AND expression AST node

Return type
ExprNode

parse_atom() → tuple[Literal['NAME'], str] | tuple[Literal['NOT'], ExprNode] | tuple[Literal['AND', 'OR'], ExprNode, ExprNode]

Parse an atomic expression (NAME or parenthesised expression).

Returns
The parsed atomic expression AST node

Return type
ExprNode

Raises
ParseError – If an unexpected token is encountered

parse_not() → tuple[Literal['NAME'], str] | tuple[Literal['NOT'], ExprNode] | tuple[Literal['AND', 'OR'], ExprNode, ExprNode]

Parse a NOT expression.

Returns
The parsed NOT expression AST node

Return type
ExprNode

parse_or() → tuple[Literal['NAME'], str] | tuple[Literal['NOT'], ExprNode] | tuple[Literal['AND', 'OR'], ExprNode, ExprNode]

Parse an OR expression.

Returns
The parsed OR expression AST node

Return type
ExprNode

peek() → tuple[str, str] | None

Peek at the next token without consuming it.

Returns
The next token if available, otherwise None

Return type
Token | None

class mafw.db.db_filter.**FilterNode**

Bases: object

Abstract base for nodes.

class mafw.db.db_filter.**LogicalNode**(*op: str*, **children:* FilterNode)

Bases: *FilterNode*

Logical combination of child nodes.

This class represents logical operations (AND, OR, NOT) applied to filter nodes. It enables building complex filter expressions by combining simpler filter nodes with logical operators.

Added in version v2.0.0.

Initialize a logical node.

Parameters

- **op** (*str*) – The logical operation ('AND', 'OR', 'NOT')
- **children** (FilterNode) – Child filter nodes to combine with the logical operation

to_expression(*model: type[Model]*) → Expression | bool

Convert this logical node to a Peewee expression.

This method evaluates the logical operation on the child nodes and returns the corresponding Peewee expression.

Parameters
model (*type[Model]*) – The model class to generate the expression for

Returns
A Peewee expression representing this logical node

Return type
peewee.Expression | bool

Raises
ValueError – If an unknown logical operation is specified

class mafw.db.db_filter.**ModelFilter**(*name_: str*, ***kwargs: Any*)

Bases: object

Class to filter rows from a model.

The filter object can be used to generate a where clause to be applied to Model.select().

The construction of a ModelFilter is normally done via a configuration file using the *from_conf()* class method. The name of the filter is playing a key role in this. If it follows a dot structure like:

ProcessorName.__filter__.ModelName

then the corresponding table from the TOML configuration object will be used.

For each processor, there might be many Filters, up to one for each Model used to get the input list. If a processor is joining together three Models when performing the input select, there will be up to three Filters collaborating on making the selection.

The filter configuration can contain the following key, value pair:

- key / string pairs, where the key is the name of a field in the corresponding Model
- key / numeric pairs
- key / arrays
- key / dict pairs with 'op' and 'value' keys for explicit operation specification

All fields from the configuration file will be added to the instance namespace, thus accessible with the dot notation. Moreover, the field names and their filter value will be added to a private dictionary to simplify the generation of the filter SQL code.

The user can use the filter object to store selection criteria. He can construct queries using the filter contents in the same way as he could use processor parameters.

If he wants to automatically generate valid filtering expression, he can use the *filter()* method. In order for this to work, the ModelFilter object be *bound* to a Model. Without this binding the ModelFilter will not be able to automatically generate expressions.

For each field in the filter, one condition will be generated according to the following scheme:

Filter field type	Logical operation	Example
Numeric, boolean	==	Field == 3.14
String	GLOB	Field GLOB '*ree'
List	IN	Field IN [1, 2, 3]
Dict (explicit)	op from dict	Field BIT_AND 5

All conditions will be joined with a AND logic by default, but this can be changed.

The ModelFilter also supports logical expressions to combine multiple filter conditions using AND, OR, and NOT operators. These expressions can reference named filter conditions within the same filter or even combine conditions from different filters when used with *ProcessorFilter*.

Conditional filters allow expressing logic like: "IF field_a IN [x, y] THEN field_b IN [1, 2] ELSE no constraint on field_b"

Consider the following example:

```
class MeasModel(MAFwBaseModel):
    meas_id = AutoField(primary_key=True)
    sample_name = TextField()
    successful = BooleanField()
    flags = IntegerField()
    composite_image_id = IntegerField()
    sample_id = IntegerField()


# Traditional simplified usage
flt = ModelFilter(
    'MyProcessor.__filter__.MyModel',
    sample_name='sample_00*',
    meas_id=[1, 2, 3],
    successful=True,
)

# New explicit operation usage
flt = ModelFilter(
    'MyProcessor.__filter__.MyModel',
    sample_name={'op': 'LIKE', 'value': 'sample_00%'},
    flags={'op': 'BIT_AND', 'value': 5},
    meas_id={'op': 'IN', 'value': [1, 2, 3]},
)

# Logical expression usage
flt = ModelFilter(
    'MyProcessor.__filter__.MyModel',
    sample_name={'op': 'LIKE', 'value': 'sample_00%'},
```

(continues on next page)

(continued from previous page)

```
    flags={'op': 'BIT_AND', 'value': 5},
    meas_id={'op': 'IN', 'value': [1, 2, 3]},
    __logic__='sample_name AND (flags OR meas_id)',
)

# Conditional filter usage
flt = ModelFilter(
    'MyProcessor.__filter__.MyModel',
    sample_name='sample_00*',
    composite_image_id=[100, 101],
    sample_id=[1, 2],
    __conditional__=[
        {
            'condition_field': 'composite_image_id',
            'condition_op': 'IN',
            'condition_value': [100, 101],
            'then_field': 'sample_id',
            'then_op': 'IN',
            'then_value': [1, 2],
        }
    ],
)

flt.bind(MeasModel)
filtered_query = MeasModel.select().where(flt.filter())
```

The explicit operation format allows for bitwise operations and other advanced filtering.

TOML Configuration Examples:

```
[MyProcessor.__filter__.MyModel]
sample_name = "sample_00*"  # Traditional GLOB
successful = true           # Traditional equality

# Explicit operations
flags = { op = "BIT_AND", value = 5 }
score = { op = ">=", value = 75.0 }
category = { op = "IN", value = ["A", "B", "C"] }
date_range = { op = "BETWEEN", value = ["2024-01-01", "2024-12-31"] }

# Logical expression for combining conditions
__logic__ = "sample_name AND (successful OR flags)"

# Conditional filters
[[MyProcessor.__filter__.MyModel.__conditional__]]
condition_field = "composite_image_id"
condition_op = "IN"
condition_value = [100, 101]
then_field = "sample_id"
then_op = "IN"
then_value = [1, 2]

# Nested conditions with logical expressions
[MyProcessor.__filter__.MyModel.nested_conditions]
__logic__ = "a OR b"
a = { op = "LIKE", value = "test%" }
b = { op = "IN", value = [1, 2, 3] }
```

See also

- *mafw.db.db_filter.ProcessorFilter* - For combining multiple ModelFilters with logical expressions
- *mafw.db.db_filter.ConditionalFilterCondition* - For conditional filtering logic
- *mafw.db.db_filter.ExprParser* - For parsing logical expressions

Constructor parameters:

Parameters

- **name_** (*str*) – The name of the filter. It should be in dotted format to facilitate the configuration via the steering file. The _ is used to allow the user to have a keyword argument named name.
- **kwargs** – Keyword parameters corresponding to fields and filter values.

Changed in version v1.2.0: The parameter *name* has been renamed as *name_*.

Changed in version v1.3.0: Implementation of explicit operation.

Changed in version v2.0.0: Introduction of conditional filters, logical expression and hierarchical structure. Introduction of autobinding for MAFwBaseModels

classmethod from_conf(*name: str*, *conf: dict[str, Any]*) → Self

Builds a Filter object from a steering file dictionary.

If the name is in dotted notation, then this should be corresponding to the table in the configuration file. If a default configuration is provided, this will be used as a starting point for the filter, and it will be updated by the actual configuration in `conf`.

In normal use, you would provide the specific configuration via the conf parameter.

See details in the *class documentation*

Parameters

- **name** (*str*) – The name of the filter in dotted notation.
- **conf** (*dict*) – The configuration dictionary.

Returns
A Filter object

Return type
ModelFilter

static _create_condition_node_from_value(*value: Any*, *field_name: str*, *node_name: str | None = None*) → *ConditionNode*

Create a FilterCondition based on value type (backward compatibility).

Parameters

- **value** – The filter value
- **field_name** – The field name

Returns
A FilterCondition

_auto_bind() → None

Attempt to automatically bind the filter to a model.

This method tries to retrieve the model associated with the filter's model name from the *mafw.db.db_model* registry and bind it using the *bind()* method.

If the model cannot be found,a warning is logged indicating the failure to perform auto-binding.

This model is automatically invoked by the `ModelFilter` constructor.

_build_logical_node_from_ast(*ast: tuple[Literal['NAME'], str] | tuple[Literal['NOT'], tuple[Literal['NAME'], str] | tuple[Literal['NOT'], ExprNode] | tuple[Literal['AND', 'OR'], ExprNode, ExprNode]] | tuple[Literal['AND', 'OR'], tuple[Literal['NAME'], str] | tuple[Literal['NOT'], ExprNode] | tuple[Literal['AND', 'OR'], ExprNode, ExprNode], tuple[Literal['NAME'], str] | tuple[Literal['NOT'], ExprNode] | tuple[Literal['AND', 'OR'], ExprNode, ExprNode]]*, *name_to_nodes: dict[str,* FilterNode*]*, *model_name_placeholder: str | None = None*) → *FilterNode*

Recursively build LogicalNode from AST using a mapping name->FilterNode.

_evaluate_logic_ast(*ast: tuple[Literal['NAME'], str] | tuple[Literal['NOT'], tuple[Literal['NAME'], str] | tuple[Literal['NOT'], ExprNode] | tuple[Literal['AND', 'OR'], ExprNode, ExprNode]] | tuple[Literal['AND', 'OR'], tuple[Literal['NAME'], str] | tuple[Literal['NOT'], ExprNode] | tuple[Literal['AND', 'OR'], ExprNode, ExprNode], tuple[Literal['NAME'], str] | tuple[Literal['NOT'], ExprNode] | tuple[Literal['AND', 'OR'], ExprNode, ExprNode]]*) → Expression | bool

Evaluate an abstract syntax tree (AST) representing a logical expression.

This method recursively evaluates the AST nodes to produce a Peewee expression or boolean value representing the logical combination of filter conditions.

Parameters
ast (*Any*) – The abstract syntax tree node to evaluate

Returns
A Peewee expression for logical operations or boolean True/False

Return type
peewee.Expression | bool

Raises

- **KeyError** – If a referenced condition name is not found in the filter
- **ValueError** – If an unsupported AST node type is encountered

add_conditional(*conditional:* ConditionalFilterCondition) → None

Add a conditional filter.

Added in version v2.0.0.

Parameters
conditional (`ConditionalFilterCondition`) – The conditional filter condition

add_conditional_from_dict(*config: dict[str, Any]*) → None

Add a conditional filter from a configuration dictionary.

Added in version v2.0.0.

Parameters
config (*dict[str, Any]*) – Dictionary with conditional filter configuration

bind(*model: type[Model]*) → None

Connects a filter to a Model class.

Parameters
model (*Model*) – Model to be bound.

filter(*join_with: Literal['AND', 'OR'] = 'AND'*) → Expression | bool

Generates a filtering expression joining all filtering fields.

See details in the *class documentation*

Changed in version v1.3.0: Add the possibility to specify a *join_with* function

Changed in version v2.0.0: Add support for conditional filters and for logical expression

Parameters

join_with (*Literal['AND', 'OR'], default 'AND'*) – How to join conditions ('AND' or 'OR'). Defaults to 'AND'.

Returns

The filtering expression.

Return type

peewee.Expression | bool

Raises

- **TypeError** – when the field value type is not supported.
- **ValueError** – when join_with is not 'AND' or 'OR'.

conditional_name = '__conditional__'

The conditional keyword identifier.

This value cannot be used as field name in the filter bound model.

property is_bound: bool

Returns true if the ModelFilter has been bound to a Model

logic_name = '__logic__'

The logic keyword identifier.

This value cannot be used as field name in the filter bound model.

class mafw.db.db_filter.**ProcessorFilter**(*data: dict[str, ModelFilter] | None = None, /, *(Positional-only parameter separator (PEP 570))*, **kwargs: Any*)

Bases: UserDict[str, *ModelFilter*]

A special dictionary to store all *Filters* in a processors.

It contains a publicly accessible dictionary with the configuration of each ModelFilter using the Model name as keyword.

It contains a private dictionary with the global filter configuration as well. The global filter is not directly accessible, but only some of its members will be exposed via properties. In particular, the new_only flag that is relevant only at the Processor level can be accessed directly using the *new_only*. If not specified in the configuration file, the new_only is by default True.

It is possible to assign a logic operation string to the register that is used to join all the filters together when performing the *filter_all()*. If no logic operation string is provided, the register will provide a join condition using either AND (default) or OR.

Constructor parameters:

Parameters

- **data** (*dict*) – Initial data
- **kwargs** – Keywords arguments

bind_all(*models: list[type[Model]] | dict[str, type[Model]]*) → None

Binds all filters to their models.

The `models` list or dictionary should contain a valid model for all the ModelFilters in the registry. In the case of a dictionary, the key value should be the model name.

Parameters
models (*list[type(Model)] | dict[str,type(Model)]*) – List or dictionary of a databank of Models from which the ModelFilter can be bound.

filter_all(*join_with: Literal['AND', 'OR'] = 'AND'*) → Expression | bool

Generates a where clause joining all filters.

If a logic expression is present, it will be used to combine named filters. Otherwise, fall back to the legacy behaviour using join_with.

Raises
ValueError – If the parsing of the logical expression fails

Parameters
join_with (*Literal['AND', 'OR'], default: 'AND'*) – Logical function to join the filters if no logic expression is provided.

Returns
ModelFilter expression

Return type
peewee.Expression

property new_only: bool

The new only flag.

Returns
True, if only new items, not already in the output database table must be processed.

Return type
bool

mafw.db.db_filter.**_format_expected**(*expected: str | Sequence[str]*) → str

Format a description of expected tokens for diagnostics.

mafw.db.db_filter.**_tokenize_with_positions**(*text: str*) → tuple[list[tuple[str, str]], list[int]]

Tokenize text while capturing the start offset of each token.

mafw.db.db_filter.**ast_to_string**(*ast: tuple[Literal['NAME'], str] | tuple[Literal['NOT'], tuple[Literal['NAME'], str] | tuple[Literal['NOT'], ExprNode] | tuple[Literal['AND', 'OR'], ExprNode, ExprNode]] | tuple[Literal['AND', 'OR'], tuple[Literal['NAME'], str] | tuple[Literal['NOT'], ExprNode] | tuple[Literal['AND', 'OR'], ExprNode, ExprNode], tuple[Literal['NAME'], str] | tuple[Literal['NOT'], ExprNode] | tuple[Literal['AND', 'OR'], ExprNode, ExprNode]]*) → str

Convert an abstract syntax tree (AST) back to its string representation.

Parameters
ast (*ExprNode*) – The AST to convert

Returns
The string representation of the AST

Return type
str

mafw.db.db_filter.**tokenize**(*text: str*) → list[tuple[str, str]]

Tokenize a logical expression string into a list of tokens.

This function breaks down a logical expression string into individual tokens based on the defined token specifications. It skips whitespace and raises a *ParseError* for unexpected characters.

Parameters
text (*str*) – The logical expression string to tokenize

Returns
A list of tokens represented as (token_type, token_value) tuples

Return type
list[*Token*]

Raises
ParseError – If an unexpected character is encountered in the text

mafw.db.db_filter.**BinaryNode**

AND/OR nodes are tuples of the operator and two recursive nodes

alias of tuple[Literal['AND', 'OR'], ExprNode, ExprNode]

mafw.db.db_filter.**ExprNode: TypeAlias = tuple[typing.Literal['NAME'], str] | tuple[typing.Literal['NOT'], 'ExprNode'] | tuple[typing.Literal['AND', 'OR'], 'ExprNode', 'ExprNode']**

The main recursive type combining all options

This type represents the abstract syntax tree (AST) nodes used in logical expressions. It can be one of:

- *NameNode*: A named element (field name or filter name)
- *NotNode*: A negation operation
- *BinaryNode*: An AND/OR operation between two nodes

mafw.db.db_filter.**MASTER_RE = re.compile('(?P<LPAREN>\\()|(?P<RPAREN>\\))|(?P<AND>\\bAND\\b)|(?P<OR>\\bOR\\b)|(?P<NOT>\\bNOT\\b)|(?P<NAME>[A-Za-z_][A-Za-z0-9_\\.]*(?:\\:[A-Za-z_][A-Za-z0-9_]*)?)|(?P<SKIP>[\\t\\n\\r]+)|(?P<MISMATCH>.)')**

Compiled regular expression to interpret the logical expression grammar

mafw.db.db_filter.**NameNode**

An atom is a tuple of the literal string 'NAME' and the value

alias of tuple[Literal['NAME'], str]

mafw.db.db_filter.**NotNode**

A NOT node is a tuple of 'NOT' and a recursive node

alias of tuple[Literal['NOT'], ExprNode]

mafw.db.db_filter.**TOKEN_SPECIFICATION = [('LPAREN', '\\('), ('RPAREN', '\\)'), ('AND', '\\bAND\\b'), ('OR', '\\bOR\\b'), ('NOT', '\\bNOT\\b'), ('NAME', '[A-Za-z_][A-Za-z0-9_\\.]*(?:\\:[A-Za-z_][A-Za-z0-9_]*)?'), ('SKIP', '[\\t\\n\\r]+'), ('MISMATCH', '.')]**

Token specifications

mafw.db.db_filter.**Token**

Type definition for a logical expression token

alias of tuple[str, str]

mafw.db.db_model

The module provides functionality to MAFw to interface to a DB.

Module Attributes

database_proxy	This is a placeholder for the real database object that will be known only at run time
mafw_model_register	This is the instance of the ModelRegister

Functions

get_db(model_class)	Extract the actual database instance from a model class.
get_primary_key_fields(model_class)	Extract primary key field names and field objects from a model metadata class.
make_prefixed_suffixed_name(model_class)	Generate a table name with optional prefix and suffix for a given model class.

Classes

MAFwBaseModel(*args, **kwargs)	The base model for the MAFw library.
RegisteredMeta(name, bases, attrs, **kwargs)	Metaclass for registering models with the MAFw model registry.

Exceptions

MAFwBaseModelDoesNotExist	Raised when the base model class is not existing.

exception mafw.db.db_model.**MAFwBaseModelDoesNotExist**

Bases: *MAFwException*

Raised when the base model class is not existing.

class mafw.db.db_model.**MAFwBaseModel**(**args*, ***kwargs*)

Bases: Model

The base model for the MAFw library.

Every model class (table) that the user wants to interface must inherit from this base.

This class extends peewee's Model with several additional features:

1. Automatic model registration: Models are automatically registered with the MAFw model registry during class definition, enabling dynamic discovery and management of database models.
2. Trigger support: The class supports defining database triggers through the *triggers()* method, which are automatically created when the table is created. File removal triggers can also be automatically generated using the *file_trigger_auto_create* boolean flag in the *meta class*. See also *file_removal_triggers()*.
3. Standard upsert operations: Provides *std_upsert()* and *std_upsert_many()* methods for performing upsert operations that work with SQLite and PostgreSQL.
4. Dictionary conversion utilities: Includes *to_dict()*, *from_dict()*, and *update_from_dict()* methods for easy serialization and deserialization of model instances.

5. Customizable table naming: Supports table name prefixes and suffixes through the Meta class with *prefix* and *suffix* attributes. See *make_prefixed_suffixed_name()*.
6. Automatic table creation control: The *automatic_creation* Meta attribute controls whether tables are automatically created when the application starts.

Note

The automatic model registration can be disabled for one single model class using the keyword argument *do_not_register* passed to the *RegisteredMeta* meta-class. For example:

```
class AutoRegisterModel(MAFwBaseModel):
    pass


class NoRegisterModel(MAFwBaseModel, do_not_register=True):
    pass
```

the first class will be automatically registered, while the second one will not. This is particularly useful if the user wants to define a base model class for the whole project without having it in the register where only concrete Model implementations are stored.

DoesNotExist

alias of *MAFwBaseModelDoesNotExist*

classmethod create_table(*safe: bool = True*, ***options: Any*) → None

Create the table in the underlying DB and all the related trigger as well.

If the creation of a trigger fails, then the whole table dropped, and the original exception is re-raised.

Warning

Trigger creation has been extensively tested with SQLite, but not with the other database implementation. Please report any malfunction.

Parameters

- **safe** (*bool, Optional*) – Flag to add an IF NOT EXISTS to the creation statement. Defaults to True.
- **options** – Additional options passed to the super method.

classmethod file_removal_triggers() → list[*Trigger*]

Generate a list of triggers for automatic file removal when records are deleted.

This method creates database triggers that automatically handle file cleanup when records containing *FileNameField* fields are removed from the database table. The triggers insert the filenames and checksums into the *OrphanFile* table for later processing.

The triggers are only created if the model has at least one field of type *FileNameField*. If no such fields exist, an empty list is returned.

FileNameListField is a subclass of *FileNameField* and is treated in the same way.

Added in version v2.0.0.

Note

This functionality requires the `file_trigger_auto_create` attribute in the model's Meta class to be set to `True` for automatic trigger creation.

Returns
A list containing the trigger object for file removal, or an empty list if no *FileNameField* fields are found.

Return type
list[*Trigger*]

classmethod from_dict(*data: dict[str, Any]*, *ignore_unknown: bool = False*) → *MAFwBaseModel*

Create a new model instance from dictionary

Parameters

- **data** (*dict[str, Any]*) – The dictionary containing the key/value pairs of the model.
- **ignore_unknown** (*bool*) – If unknown dictionary keys should be ignored.

Returns
A new model instance.

Return type
MAFwBaseModel

classmethod get_fields_by_type(*field_type: type[Field]*) → dict[str, Field]

Return a dict {field_name: field_object} for all fields of the given type.

Added in version v2.0.0.

Parameters
field_type (*peewee.Field*) – Field type

Returns
A dict {field_name: field_object} for all fields of the given type.

Return type
dict[str, peewee.Field]

classmethod join_dependent_models(*base_query: Select | None = None*) → Select

Return a query that joins every model referenced through foreign keys.

Each foreign-key relation defined on `cls` is joined and attached to the resulting rows via an attribute named `_<field name>` so that callers have the same convenient access that manual `.join_from` calls would have provided.

Parameters
base_query (`peewee.Select`, optional) – Optional query to extend. If not provided a new `Select` is created that includes `cls` and the dependent models.

Returns
The query joining every dependent model.

Return type
`peewee.Select`

classmethod std_upsert(*_MAFwBaseModel__data: dict[str, Any] | None = None*, ***mapping: Any*) → ModelInsert

Perform a so-called standard upsert.

An upsert statement is not part of the standard SQL and different databases have different ways to implement it. This method will work for the three DB backends generating a valid query for the three dialects.

An upsert is a statement in which we try to insert some data in a table where there are some constraints. If one constraint is failing, then instead of inserting a new row, we will try to update the existing row causing the constraint violation.

A standard upsert, in the naming convention of MAFw, is setting the conflict cause to the primary key with all other fields being updated. In other words, the database will try to insert the data provided in the table, but if the primary key already exists, then all other fields will be updated.

This method is equivalent to the following:

```
class Sample(MAFwBaseModel):
    sample_id = AutoField(
        primary_key=True,
        help_text='The sample id primary key',
    )
    sample_name = TextField(help_text='The sample name')


(
    Sample.insert(sample_id=1, sample_name='my_sample')
    .on_conflict(
        preserve=[Sample.sample_name]
    )  # use the value we would have inserted
    .execute()
)
```

Parameters

- **__data** (*dict, Optional*) – A dictionary containing the key/value pair for the insert. The key is the column name. Defaults to None
- **mapping** – Keyword arguments representing the value to be inserted.

classmethod std_upsert_many(*rows: Iterable[Any]*, *fields: list[str] | None = None*) → ModelInsert

Perform a standard upsert with many rows.

See also

Read the *std_upsert()* documentation for an explanation of this method.

Parameters

- **rows** (*Iterable*) – A list with the rows to be inserted. Each item can be a dictionary or a tuple of values. If a tuple is provided, then the *fields* must be provided.
- **fields** (*list[str], Optional*) – A list of field names. Defaults to None.

classmethod triggers() → list[*Trigger*]

Returns an iterable of *Trigger* objects to create upon table creation.

The user must overload this returning all the triggers that must be created along with this class.

to_dict(*recurse: bool = True*, *backrefs: bool = False*, *only: list[str] | None = None*, *exclude: list[str] | None = None*, ***kwargs: Any*) → dict[str, Any]

Convert model instance to dictionary with optional parameters

See full documentation directly on the peewee documentation.

Parameters

- **recurse** (*bool, Optional*) – If to recurse through foreign keys. Default to True.
- **backrefs** (*bool, Optional*) – If to include backrefs. Default to False.
- **only** (*list[str], Optional*) – A list of fields to be included. Defaults to None.

- **exclude** (*list[str], Optional*) – A list of fields to be excluded. Defaults to None.
- **kwargs** – Other keyword arguments to be passed to peewee playhouse shortcut.

Returns
A dictionary containing the key/value of the model.

Return type
dict[str, Any]

update_from_dict(*data: dict[str, Any]*, *ignore_unknown: bool = False*) → *MAFwBaseModel*

Update current model instance from dictionary

The model instance is returned for daisy-chaining.

Parameters

- **data** (*dict[str, Any]*) – The dictionary containing the key/value pairs of the model.
- **ignore_unknown** (*bool*) – If unknown dictionary keys should be ignored.

class mafw.db.db_model.**RegisteredMeta**(*name: str*, *bases: tuple[type, ...]*, *attrs: dict[str, Any]*, ***kwargs: dict[str, Any]*)

Bases: ModelBase

Metaclass for registering models with the MAFw model registry.

This metaclass automatically registers model classes with the global model registry when they are defined, allowing for dynamic discovery and management of database models. It ensures that only concrete model classes (not the base classes themselves) are registered.

The registration process uses the table name from the model's metadata or generates a snake_case version of the class name if no explicit table name is set.

Create a new model class and register it if applicable.

This method is called during class definition to create the actual class and register it with the MAFw model registry if it's a concrete model class.

Parameters

- **name** (*str*) – The name of the class being created.
- **bases** (*tuple*) – The base classes of the class being created.
- **attrs** (*dict*) – The attributes of the class being created.
- **kwargs** (*dict[str, Any]*) – Other keyword attributes passed to the class.

Returns
The newly created class.

Return type
type

mafw.db.db_model.**get_db**(*model_class:* PeeweeModelWithMeta) → Database

Extract the actual database instance from a model class.

This function retrieves the database object from a model class's metadata, handling the case where the database is a peewee.DatabaseProxy. If the database is a proxy, it extracts the underlying database object.

Parameters
model_class (PeeweeModelWithMeta) – The model class from which to extract the database.

Returns
The actual database instance.

Return type
Database

mafw.db.db_model.**get_primary_key_fields**(*model_class:* PeeweeModelWithMeta) → tuple[list[str], list[Field]]

Extract primary key field names and field objects from a model metadata class.

This function retrieves the primary key information from a Peewee model metadata class, handling both single-column primary keys and composite (multi-column) keys. It returns two parallel lists: the first containing the field names as strings, and the second containing the corresponding field objects.

Parameters
model_class (RegisteredMeta) – The model class from which to extract primary key information.

Returns
A tuple containing two lists: the first with primary key field names (as strings) and the second with the corresponding field objects.

Return type
tuple[list[str], list[Field]]

mafw.db.db_model.**make_prefixed_suffixed_name**(*model_class:* RegisteredMeta) → str

Generate a table name with optional prefix and suffix for a given model class.

This function constructs a table name by combining the prefix, the snake_case version of the model class name, and the suffix. If either prefix or suffix are not defined in the model's metadata, empty strings are used instead.

The prefix, table name, and suffix are joined using underscores. For example:

- If a model class is named "UserAccount" with prefix="app", suffix="data", the resulting table name will be "app_user_account_data"
- If a model class is named "Product" with prefix="ecommerce", suffix="_latest", the resulting table name will be "ecommerce_product_latest"

Note

Underscores (_) will be automatically added to prefix and suffix if not already present.

Parameters
model_class (RegisteredMeta) – The model class for which to generate the table name.

Returns
The constructed table name including prefix and suffix if applicable.

Return type
str

mafw.db.db_model.**database_proxy = <peewee.DatabaseProxy object>**

This is a placeholder for the real database object that will be known only at run time

mafw.db.db_model.**mafw_model_register = <mafw.db.model_register.ModelRegister object>**

This is the instance of the ModelRegister

See also

ModelRegister for more information on how to retrieve models and *RegisteredMeta* and *MAFwBaseModel* for the automatic registration of models`

mafw.db.db_types

Database Type Definitions

This module provides type definitions for database models and related components.

It defines Protocol classes that represent the expected interfaces for database models, helping with static type checking when working with ORM frameworks like Peewee.

Classes

PeeweeModelWithMeta(*args, **kwargs)	Protocol defining the interface for Peewee model classes with metadata.

class mafw.db.db_types.**PeeweeModelWithMeta**(**args*, ***kwargs*)

Bases: `Protocol`

Protocol defining the interface for Peewee model classes with metadata.

This Protocol helps with static type checking for Peewee ORM models, ensuring that objects passed to functions expecting Peewee models have the necessary methods and attributes.

Attributes

_meta : Any
The metadata object associated with the Peewee model class. Contains information about the table, fields, and other model properties.

Methods

select(*args: Any, **kwargs: Any) -> Any
Select records from the database.

delete(*args: Any, **kwargs: Any) -> Any
Delete records from the database.

mafw.db.db_wizard

The module allows to generate a DB model structure starting from an existing DB.

It is strongly based on the peewee playhouse module *pwiz*.

Functions

dump_models(output_file, introspector[, ...])	Dumps all the ORM models in the output file.

Classes

UnknownField(*_, **__)	Placeholder class for an Unknown Field.

class mafw.db.db_wizard.**UnknownField**(**_: Any*, ***__: Any*)

Bases: `object`

Placeholder class for an Unknown Field.

mafw.db.db_wizard.**dump_models**(*output_file: TextIO*, *introspector: Introspector*, *tables: list[str] | tuple[str, ...] | None = None*, *preserve_order: bool = True*, *include_views: bool = False*, *ignore_unknown: bool = False*, *snake_case: bool = True*) → None

Dumps all the ORM models in the output file.

This function will write to the output stream a fully functional python module with all the models that the introspector class can access. The user has the possibility to pre-select a subset of tables to be dumped and also to optionally include views.

See also

This function is the core of the db wizard.

Parameters

- **output_file** (*TextIO*) – The output file. It must be open in text/write mode.
- **introspector** (*Introspector*) – A database introspector from the reflection module.
- **tables** (*list | None, Optional*) – A list of table names to be dumped. If None, then all tables in the DB will be dumped. Defaults to None.
- **preserve_order** (*bool, Optional*) – Preserve the order of the columns in the model. Defaults to True.
- **include_views** (*bool, Optional*) – Include saved views to be dumped. Views can be obtained joining tables. Defaults to False.
- **ignore_unknown** (*bool, Optional*) – Ignore unknown fields. If True, then an UnknownField type will be used as a placeholder. Defaults to False.
- **snake_case** (*bool, Optional*) – Use snake case for column and table name. Defaults to True.

mafw.db.fields

Module provides customised model fields specific for MAFw.

Classes

FileChecksumField([null, index, unique, ...])	A field to be used for file checksum.
FileChecksumFieldAccessor(model, field, name)	Field accessor specialized for file checksum fields.
FileNameField([checksum_field])	Field to be used for filenames.
FileNameFieldAccessor(model, field, name)	A field accessor specialized for filename fields.
FileNameListField([checksum_field])	A field for a list of file names.

class mafw.db.fields.**FileChecksumField**(*null=False*, *index=False*, *unique=False*, *column_name=None*, *default=None*, *primary_key=False*, *constraints=None*, *sequence=None*, *collation=None*, *unindexed=False*, *choices=None*, *help_text=None*, *verbose_name=None*, *index_type=None*, *db_column=None*, *_hidden=False*)

Bases: TextField

A field to be used for file checksum.

It is the evolution of the TextField for storing file checksum hexadecimal digest.

If linked to a *FileNameField* or *FileNameListField*, then it will be automatically filled when the primary file name (or list of file names) field is set.

If the user decides to set its value manually, then he can provide either the string with the hexadecimal characters as calculated by *file_checksum()*, or simply the filename (or filename list) and the field will perform the calculation automatically.

accessor_class

The specific field accessor class

alias of *FileChecksumFieldAccessor*

db_value(*value: str | Path | list[str | Path]*) → str

Converts the python assigned value to the DB type.

The checksum will be stored in the DB as a string containing only hexadecimal characters (see hexdigest).

The user can provide the checksum directly or the path to the file or a list of path to files. If a Path, or list of Path, is provided, then the checksum will be calculated, if a str (a path converted into a string) is provided, the function will try to see if a file with that path exists. If so, the checksum will be calculated, if not, the original string is assumed to be the checksum.

Parameters
value (*str | Path | list[str | Path]*) – The checksum or path to the file, or list of path to files for which the checksum has to be stored.

Returns
The checksum string for the database storage.

Return type
str

class mafw.db.fields.**FileChecksumFieldAccessor**(*model*, *field*, *name*)

Bases: FieldAccessor

Field accessor specialized for file checksum fields.

When the field is directly set, then an initialization flag in the model instance is turned to True to avoid that the linked primary field will overrule this value again.

For each checksum field named my_checksum, the model instance will get an attribute: init_my_checksum to be used as an initialization flag.

Once the field is manually set, to re-establish the automatic mechanism, the user has to manually toggle the initialization flag.

class mafw.db.fields.**FileNameField**(*checksum_field: str | None = None*, **args: Any*, ***kwargs: Any*)

Bases: TextField

Field to be used for filenames.

It is just an overload of TextField, that allows to apply filters and python functions specific to filenames.

If the user specifies the name of a file checksum field, then when this field is updated, the checksum one will also be automatically updated.

See also

- *FileNameListField* for a field able to store a list of filenames.
- *remove_widow_db_rows()* for a function removing entries from a database table where the corresponding files on disk are missing.
- *verify_checksum()* for a function comparing the actual checksum with the stored one and in case removing outdated entries from the DB.

Constructor parameter:

Parameters
checksum_field (*str, Optional*) – The name of the checksum field linked to this filename. Defaults to None.

accessor_class

The specific accessor class

alias of *FileNameFieldAccessor*

db_value(*value: str | Path*) → str

Converts the input python value into a string for the DB.

python_value(*value: str*) → Path | None

Converts the db value from str to Path

The return value might also be None, if the user set the field value to null.

Parameters
value (*str*) – The value of the field as stored in the database.

Returns
The converted value as a path. It can be None, if value was stored as null.

Return type
Path | None

class mafw.db.fields.**FileNameFieldAccessor**(*model*, *field*, *name*)

Bases: FieldAccessor

A field accessor specialized for filename fields.

In the constructor of the *FileNameField* and subclasses, the user can specify the name of a checksum field linked to this filename. This is very useful because in this way, the user does not have to manually assign any value to this field that will simply be automatically updated when the filename field is updated.

The user can disable this automatic feature either removing the link in the *FileNameField* or simply assigning a value to the *FileChecksumField*.

class mafw.db.fields.**FileNameListField**(*checksum_field: str | None = None*, **args: Any*, ***kwargs: Any*)

Bases: *FileNameField*

A field for a list of file names.

The evolution of the *FileNameField*, this field is able to store a list of filenames as a ';' separated string of full paths.

It is meant to be used when a processor is saving a bunch of correlated files that are to be used all together.

In a similar way as its parent class, it can be link to a checksum field, in this case, the checksum of the whole file list will be calculated.

Constructor parameter:

Parameters
checksum_field (*str, Optional*) – The name of the checksum field linked to this filename. Defaults to None.

db_value(*value: list[str | Path] | str | Path*) → str

Converts the list of paths in a ';' separated string

python_value(*value: str*) → list[Path]

Converts the ';' separated string in a list of paths

mafw.db.model_register

Module for registering and retrieving database models.

This module provides functionality to register database models with table names as key and retrieve them using flexible naming conventions that support prefixes and suffixes.

Added in version v2.0.0.

Classes

ModelRegister()	A registry for database models with support for prefixes and suffixes.

class mafw.db.model_register.**ModelRegister**

Bases: object

A registry for database models with support for prefixes and suffixes.

This class allows registration of database models with table names and provides flexible retrieval mechanisms that can handle different naming conventions.

The following implemented methods are thread-safe:

- *register_model()*
- *register_prefix()*
- *register_suffix()*
- *get_model()*
- *get_model_names()*
- *get_table_names()*
- *items()*

Accessing the class underling containers is instead **not** thread safe.

Added in version v2.0.0.

clear() → None

Clear all registered models, prefixes, and suffixes from the registry.

This method removes all entries from the internal dictionaries and lists, effectively resetting the ModelRegister to its initial empty state.

> **Note**
>
> This operation cannot be undone. All previously registered models and naming conventions will be lost after calling this method.

Return type
None

get_model(*name: str*) → ModelBase

Retrieve a model by name, supporting prefixes and suffixes.

name could be either the table_name or the ModelName.

This method attempts to find a model by the given name, considering registered prefixes and suffixes. It also handles conversion between CamelCase model names and snake_case table names using peewee's utility.

Parameters
name (*str*) – The name to search for

Returns
The registered peewee Model class

Return type
peewee.Model

Raises
KeyError – If no matching model is found or multiple similar models exist

get_model_names() → list[str]

Get a list of all registered model names.

Returns
List of registered model names

Return type
list[str]

get_standard_tables() → list[type[*StandardTable*]]

Retrieve all registered models that are instances of *StandardTable*.

This method filters the registered models and returns only those that inherit from the *StandardTable* base class.

This is useful for identifying and working with standard database tables that follow a specific structure or interface.

Since the introduction of the *ModelRegister*, there is no need any more for a standard table plugin hook, instead the user can use this method to retrieve all standard tables.

Returns
A list of registered model classes that are standard tables

Return type
list[peewee.ModelBase]

get_table_names() → list[str]

Get a list of all registered table names.

Returns
List of registered table names

Return type
list[str]

items() → list[tuple[str, ModelBase]]

Return the items list of the registered models dictionary.

This method provides access to all registered models through a dictionary-like items view, allowing iteration over key-value pairs of table names and their corresponding model classes.

In order to release the thread lock as soon as possible, instead of providing an iterator a list of the current snapshot of the dictionary is provided.

Returns
An items view of the registered models

Return type
list[[str, peewee.ModelBase]]

register_model(*table_name: str*, *model: ModelBase*) → None

Register a model with a specific table name.

If a model with the same table name already exists, it will be replaced with a warning message.

Parameters

- **table_name** (*str*) – The table name to register the model under
- **model** (*peewee.Model*) – The peewee Model class to register

register_prefix(*prefix: str*) → None

Register a prefix to be used when searching for models.

Parameters
prefix (*str*) – The prefix string to register

register_suffix(*suffix: str*) → None

Register a suffix to be used when searching for models.

Parameters
suffix (*str*) – The suffix string to register

mafw.db.std_tables

Module provides standard tables that are included in all database created by MAFw processor.

Standard tables are automatically created and initialized by a *Processor* or a *ProcessorList* when opening a database connection.

This means that if a processor receives a valid database object, then it will suppose that the connection was already opened somewhere else (either from a ProcessorList or a third party) and thus it is not creating the standard tables.

If a processor is constructed using a database configuration dictionary, then it will first try to open a connection to the DB, then creating all standard tables and finally executing their *StandardTable.init* method. The same apply for the Processor list.

In other words, object responsible to open the database connection is taking care also of creating the standard tables and of initializing them. If the user opens the connection and passes it to a Processor or ProcessorList, then the user is responsible to create the standard tables and to initialize them.

All standard tables must derive from the *StandardTable* to have the same interface for the initialization.

Classes

OrphanFile(*args, **kwargs)	A Model for the files to be removed from disc
PlotterOutput(*args, **kwargs)	A model for the output of the plotter processors.
StandardTable(*args, **kwargs)	A base class for tables that are generated automatically by the MAFw processor.
TriggerDisabler(trigger_type_id)	A helper tool to disable a specific type of triggers.
TriggerStatus(*args, **kwargs)	A Model for the trigger status

Exceptions

OrphanFileDoesNotExist	An exception raised when trying to access a not existing table.
PlotterOutputDoesNotExist	An exception raised when trying to access a not existing table.
StandardTableDoesNotExist	An exception raised when trying to access a not existing table.
TriggerStatusDoesNotExist	An exception raised when trying to access a not existing table.

exception mafw.db.std_tables.**OrphanFileDoesNotExist**

Bases: DoesNotExist

An exception raised when trying to access a not existing table.

exception mafw.db.std_tables.**PlotterOutputDoesNotExist**

Bases: DoesNotExist

An exception raised when trying to access a not existing table.

exception mafw.db.std_tables.**StandardTableDoesNotExist**

Bases: Exception

An exception raised when trying to access a not existing table.

exception mafw.db.std_tables.**TriggerStatusDoesNotExist**

Bases: Exception

An exception raised when trying to access a not existing table.

class mafw.db.std_tables.**OrphanFile**(**args*, ***kwargs*)

Bases: *StandardTable*

A Model for the files to be removed from disc

Changed in version v2.0.0: The checksum field is set to allow null values. The class is set not to automatically generate triggers for file removal

DoesNotExist

alias of *OrphanFileDoesNotExist*

class mafw.db.std_tables.**PlotterOutput**(**args*, ***kwargs*)

Bases: *StandardTable*

A model for the output of the plotter processors.

The model has a trigger activated on delete queries to insert filenames and checksum in the OrphanFile model via the automatic file delete trigger generation.

DoesNotExist

alias of *PlotterOutputDoesNotExist*

class mafw.db.std_tables.**StandardTable**(**args*, ***kwargs*)

Bases: *MAFwBaseModel*

A base class for tables that are generated automatically by the MAFw processor.

DoesNotExist

alias of *StandardTableDoesNotExist*

classmethod init() → None

The user must overload this method, if he wants some specific operations to be performed on the model everytime the database is connected.

class mafw.db.std_tables.**TriggerDisabler**(*trigger_type_id: int*)

Bases: object

A helper tool to disable a specific type of triggers.

Not all SQL dialects allow to temporarily disable trigger execution.

In order overcome this limitation, MAFw has introduced a practical workaround. All types of triggers are active by default but they can be temporarily disabled, by changing their status in the *TriggerStatus* table.

In order to disable the trigger execution, the user has to set the status of the corresponding status to 0 and also add a when condition to the trigger definition.

Here is an example code:

```
class MyTable(MAFwBaseModel):
    id_ = AutoField(primary_key=True)
    integer = IntegerField()
    float_num = FloatField()

    @classmethod
    def triggers(cls):
```

(continues on next page)

(continued from previous page)

```
        return [
            Trigger(
                'mytable_after_insert',
                (TriggerWhen.After, TriggerAction.Insert),
                cls,
                safe=True,
            )
            .add_sql(
                'INSERT INTO target_table (id__id, half_float_num) VALUES (NEW.
↪id_, NEW.float_num / 2)'
            )
            .add_when(
                '1 == (SELECT status FROM trigger_status WHERE trigger_type_id␣
↪== 1)'
            )
        ]
```

When you want to perform a database action with the trigger disabled, you can either use this class as context manager or call the *disable()* and *enable()* methods.

```
# as a context manager
with TriggerDisabler(trigger_type_id = 1):
    # do something without triggering any trigger of type 1.

# with the explicit methods
disabler = TriggerDisabler(1)
disabler.disable()
# do something without triggering any trigger of type 1.
disabler.enable()
```

When using the two explicit methods, the responsibility to assure that the triggers are re-enabled in on the user.

Constructor parameters:

Parameters
trigger_type_id (*int*) – the id of the trigger to be temporary disabled.

disable() → None
Disable the trigger

enable() → None
Enable the trigger

class mafw.db.std_tables.**TriggerStatus**(**args*, ***kwargs*)

Bases: *StandardTable*

A Model for the trigger status

DoesNotExist
alias of *TriggerStatusDoesNotExist*

classmethod init() → None
Resets all triggers to enable when the database connection is opened.

mafw.db.trigger

Module provides a Trigger class and related tools to create triggers in the database via the ORM.

It supports SQLite, MySQL and PostgreSQL with dialect-specific SQL generation.

Functions

*and_*(*conditions)	Concatenates conditions with logical AND.
*or_*(*conditions)	Concatenates conditions with logical OR.

Classes

MySQLDialect()	MySQL-specific trigger SQL generation.
PostgreSQLDialect()	PostgreSQL-specific trigger SQL generation.
SQLiteDialect()	SQLite-specific trigger SQL generation.
Trigger(trigger_name, trigger_type, source_table)	Trigger template wrapper for use with peewee ORM.
TriggerAction(*values)	String enumerator for the trigger action (Delete, Insert, Update)
TriggerDialect()	Abstract base class for database-specific trigger SQL generation.
TriggerWhen(*values)	String enumerator for the trigger execution time (Before, After or Instead Of)

class mafw.db.trigger.**MySQLDialect**

Bases: *TriggerDialect*

MySQL-specific trigger SQL generation.

create_trigger_sql(*trigger:* Trigger) → str

Generate MySQL trigger SQL.

drop_trigger_sql(*trigger_name: str*, *safe: bool = True*, *table_name: str | None = None*) → str

Generate MySQL drop trigger SQL.

supports_safe_create() → bool

MySQL supports IF NOT EXISTS for triggers.

supports_trigger_type(*when:* TriggerWhen, *action:* TriggerAction, *on_view: bool = False*) → bool

MySQL doesn't support INSTEAD OF triggers.

supports_update_of_columns() → bool

MySQL doesn't support column-specific UPDATE triggers.

supports_when_clause() → bool

MySQL supports conditions but through WHERE instead of WHEN.

class mafw.db.trigger.**PostgreSQLDialect**

Bases: *TriggerDialect*

PostgreSQL-specific trigger SQL generation.

_clean_sql(*sql: str*) → str

Remove RETURNING clauses from SQL statements for PostgreSQL trigger functions.

Parameters
sql – The SQL statement

Returns
SQL statement without RETURNING clause

create_trigger_sql(*trigger: Trigger*) → str

Generate PostgreSQL trigger SQL.

drop_trigger_sql(*trigger_name: str*, *safe: bool = True*, *table_name: str | None = None*) → str

Generate PostgreSQL drop trigger SQL.

supports_safe_create() → bool

PostgreSQL doesn't support IF NOT EXISTS for triggers before v14, but we implement safety differently.

supports_trigger_type(*when: TriggerWhen*, *action: TriggerAction*, *on_view: bool = False*) → bool

PostgreSQL supports INSTEAD OF only on views.

supports_update_of_columns() → bool

PostgreSQL supports column-specific UPDATE triggers.

supports_when_clause() → bool

PostgreSQL supports WHEN conditions.

class mafw.db.trigger.**SQLiteDialect**

Bases: *TriggerDialect*

SQLite-specific trigger SQL generation.

create_trigger_sql(*trigger: Trigger*) → str

Generate SQLite trigger SQL.

drop_trigger_sql(*trigger_name: str*, *safe: bool = True*, *table_name: str | None = None*) → str

Generate SQLite drop trigger SQL.

supports_safe_create() → bool

SQLite supports IF NOT EXISTS for triggers.

supports_trigger_type(*when: TriggerWhen*, *action: TriggerAction*, *on_view: bool = False*) → bool

SQLite supports all trigger types except INSTEAD OF on tables (only on views).

supports_update_of_columns() → bool

SQLite supports column-specific UPDATE triggers.

supports_when_clause() → bool

SQLite supports WHEN conditions.

class mafw.db.trigger.**Trigger**(*trigger_name: str*, *trigger_type: tuple[TriggerWhen, TriggerAction]*, *source_table: type[Model] | Model | str*, *safe: bool = False*, *for_each_row: bool = False*, *update_columns: list[str] | None = None*, *on_view: bool = False*)

Bases: `object`

Trigger template wrapper for use with peewee ORM.

Constructor parameters:

Parameters

- **trigger_name** (*str*) – The name of this trigger. It needs to be unique!
- **trigger_type** (*tuple[TriggerWhen, TriggerAction]*) – A tuple with *TriggerWhen* and *TriggerAction* to specify on which action the trigger should be invoked and if before, after or instead of.
- **source_table** (*type[Model] | Model | str*) – The table originating the trigger. It can be a model class, instance, or also the name of the table.
- **safe** (*bool, Optional*) – A boolean flag to define if in the trigger creation statement a 'IF NOT EXISTS' clause should be included. Defaults to False

- **for_each_row** (*bool, Optional*) – A boolean flag to repeat the script content for each modified row in the table. Defaults to False.
- **update_columns** (*list[str], Optional*) – A list of column names. When defining a trigger on a table update, it is possible to restrict the firing of the trigger to the cases when a subset of all columns have been updated. An column is updated also when the new value is equal to the old one. If you want to discriminate this case, use the *add_when()* method. Defaults to None.
- **on_view** (*bool, Optional*) – A boolean flag to indicate if the target is a view. This affects the support for INSTEAD OF. Defaults to False.

_compile_node(*node: Node*) → str

Compile a Peewee node to SQL using the current database context.

_compile_query(*query: Query*) → str

Compile a Peewee query using the current database dialect.

_compile_sql_statement(*statement: str | Query*) → str

Return a single SQL statement string with consistent formatting.

_compile_when_conditions() → list[str]

Compile each when condition, preserving nodes until creation time.

_get_dialect() → *TriggerDialect*

Get the appropriate dialect based on the database type.

Returns
A dialect instance

_node_to_sql(*node: Node*) → str

Convert a peewee Node (Expression, Query, etc.) to a SQL string with interpolated parameters.

This is based on peewee's internal query_to_string function for debugging/logging purposes.

Added in version v2.0.0.

Parameters
node – A peewee Node object

Returns
SQL string with parameters interpolated

_value_to_sql(*value: Any*) → str

Convert a Python value to its SQL representation.

Added in version v2.0.0.

Parameters
value – A Python value (string, int, float, None, etc.)

Returns
SQL string representation of the value

add_sql(*sql: str | Query*) → Self

Add an SQL statement to be executed by the trigger.

The `sql` can be either a string containing the sql statement, or it can be any other peewee Query.

For example:

```
# assuming you have created a trigger ...

sql = AnotherTable.insert(
    field1=some_value, field2=another_value
```

(continues on next page)

(continued from previous page)

```
)
trigger.add_sql(sql)
```

In this way the SQL code is generated with parametric placeholder if needed.

Parameters
sql (*str | peewee.Query*) – The SQL statement.

Returns
self for easy chaining

Return type
Trigger

add_when(**conditions: str | Node*) → Self

Add conditions to the *when* statements.

Conditions are logically ANDed. To have mixed *OR* and *AND* logic, use the functions *and_()* and *or_()*.

The `conditions` can be either strings containing SQL conditions, or peewee Node objects (such as Expression or Query objects).

For example:

```
# String condition
trigger.add_when("NEW.status = 'active'")

# Peewee expression
subq = TriggerStatus.select(TriggerStatus.status).where(
    TriggerStatus.trigger_type == 'DELETE_FILES'
)
trigger.add_when(Value(1) == subq)
```

Changed in version v2.0.0: The argument can also be a generic peewee Node.

Parameters
conditions (*str | peewee.Node*) – Conditions to be added with logical AND. Can be strings or peewee Node objects.

Returns
self for easy chaining

Return type
Trigger

create() → str

Generates the SQL create statement.

Returns
The trigger creation statement.

Raises

- ***MissingSQLStatement*** – if no SQL statements are provided.
- ***UnsupportedDatabaseError*** – if the trigger type is not supported by the database.

drop(*safe: bool = True*) → str

Generates the SQL drop statement.

Parameters
safe (*bool, Optional*) – If True, add an IF EXIST. Defaults to True.

Returns
The drop statement

Return type
str

`set_database`(*database: Database | DatabaseProxy*) → Self

Set the database to use for this trigger.

Parameters
database – The database instance

Returns
self for easy chaining

`property database: Database | None`

Return the actual database, unwrapping proxies.

`class mafw.db.trigger.TriggerAction`(**values*)

Bases: `StrEnum`

String enumerator for the trigger action (Delete, Insert, Update)

`static _generate_next_value_`(*name*, *start*, *count*, *last_values*)

Return the lower-cased version of the member name.

`class mafw.db.trigger.TriggerDialect`

Bases: `ABC`

Abstract base class for database-specific trigger SQL generation.

`abstractmethod create_trigger_sql`(*trigger:* Trigger) → str

Generate the SQL to create a trigger for a specific database dialect.

Parameters
trigger – The trigger object

Returns
SQL string to create the trigger

`abstractmethod drop_trigger_sql`(*trigger_name: str*, *safe: bool = True*, *table_name: str | None = None*) → str

Generate the SQL to drop a trigger for a specific database dialect.

Parameters

- **trigger_name** (*str*) – The name of the trigger to drop
- **safe** (*bool, Optional*) – If True, add an IF EXISTS clause. Defaults to True.
- **table_name** (*str, Optional*) – The name of the target table for the trigger. Defaults to None.

Returns
SQL string to drop the trigger

Return type
str

`abstractmethod supports_safe_create`() → bool

Check if the database supports IF NOT EXISTS for triggers.

Returns
True if supported, False otherwise

abstractmethod supports_trigger_type(*when:* TriggerWhen, *action:* TriggerAction, *on_view:* *bool = False*) → bool

Check if the database supports the specified trigger type.

Parameters

- **when** – When the trigger should fire (BEFORE, AFTER, INSTEAD OF)
- **action** – The action that triggers the trigger (INSERT, UPDATE, DELETE)
- **on_view** – Whether the trigger is on a view

Returns
True if supported, False otherwise

abstractmethod supports_update_of_columns() → bool

Check if the database supports column-specific UPDATE triggers.

Returns
True if supported, False otherwise

abstractmethod supports_when_clause() → bool

Check if the database supports WHEN conditions.

Returns
True if supported, False otherwise

class mafw.db.trigger.**TriggerWhen**(**values*)

Bases: StrEnum

String enumerator for the trigger execution time (Before, After or Instead Of)

static _generate_next_value_(*name*, *start*, *count*, *last_values*)

Return the lower-cased version of the member name.

mafw.db.trigger.**and_**(**conditions: str*) → str

Concatenates conditions with logical AND.

Parameters
conditions (*str*) – The condition to join.

Returns
The and-concatenated string of conditions

Return type
str

mafw.db.trigger.**or_**(**conditions: str*) → str

Concatenates conditions with logical OR.

Parameters
conditions (*str*) – The condition to join.

Returns
The or-concatenated string of conditions.

Return type
str

14.1.3 mafw.decorators

The module provides some general decorator utilities that are used in several parts of the code, and that can be reused by the user community.

Module Attributes

F	TypeVar for generic function.
P	TypeVar for generic processor.
single_loop(cls)	A decorator shortcut to define a single execution processor.
for_loop(cls)	A decorator shortcut to define a for loop execution processor.
while_loop(cls)	A decorator shortcut to define a while loop execution processor.
parallel_for_loop(cls)	A decorator shortcut to define a parallel for loop execution processor.
parallel_for_loop_with_queue(cls)	A decorator shortcut to define a parallel for loop with queue execution processor.

Functions

class_depends_on_optional(module_name[, ...])	Class decorator factory to check if module module_name is available.
database_required(cls)	Modify the processor start method to check if a database object exists.
depends_on_optional(module_name[, raise_ex, ...])	Function decorator to check if module_name is available.
execution_workflow([loop_type])	A decorator factory for the definition of the looping strategy.
for_loop(cls)	A decorator shortcut to define a for loop execution processor.
orphan_protector(cls)	A class decorator to modify the init method of a Processor so that the remove_orphan_files is set to False and no orphan files will be removed.
parallel_for_loop(cls)	A decorator shortcut to define a parallel for loop execution processor.
parallel_for_loop_with_queue(cls)	A decorator shortcut to define a parallel for loop with queue execution processor.
processor_depends_on_optional(module_name[, ...])	Class decorator factory to check if module module_name is available.
single_loop(cls)	A decorator shortcut to define a single execution processor.
singleton(cls)	Make a class a Singleton class (only one instance)
suppress_warnings(func)	Decorator to suppress warnings during the execution of a test function.
while_loop(cls)	A decorator shortcut to define a while loop execution processor.

class mafw.decorators.**F**

TypeVar for generic function.

alias of TypeVar('F', bound=`Callable`[[...], `object`])

class mafw.decorators.**P**

TypeVar for generic processor.

alias of TypeVar('P', bound=*Processor*)

mafw.decorators.**class_depends_on_optional**(*module_name: str*, *raise_ex: bool = False*, *warn: bool = True*) → Callable[[type[Any]], type[Any]]

Class decorator factory to check if module module_name is available.

It checks if all the optional modules listed in *module_name* separated by a ';' can be found.

If all modules are found, then the class is returned as it is.

If at least one module is not found:

- and raise_ex is True, an ImportError exception is raised and the user is responsible to deal with it.
- if raise_ex is False, instead of returning the class, a new empty class is returned.
- depending on the value of warn, the user will be informed with a warning message or not.

Parameters

- **module_name** (*str*) – The optional module(s) from which the class depends on. A ";" separated list of modules can also be provided.
- **raise_ex** (*bool, Optional*) – Flag to raise an exception if module_name not found, defaults to False.
- **warn** (*bool, Optional*) – Flag to display a warning message if module_name is not found, defaults to True.

Returns
The wrapped class.

Return type
type(object)

Raises
ImportError – if module_name is not found and raise_ex is True.

mafw.decorators.**database_required**(*cls*)

Modify the processor start method to check if a database object exists.

This decorator must be applied to processors requiring a database connection.

It operates on a processor class modifying its start method. Since the start method is also one of those for which the presence of the super call is verified, this decorator is also taking care of re-applying the super call wrapper.

Parameters
cls – A Processor class.

mafw.decorators.**depends_on_optional**(*module_name: str*, *raise_ex: bool = False*, *warn: bool = True*) → Callable[[*F*], Callable[[...], Any]]

Function decorator to check if module_name is available.

If module_name is found, then returns the wrapped function. If not, its behaviour depends on the raise_ex and warn_only values. If raise_ex is True, then an ImportError exception is raised. If it is False and warn is True, then a warning message is displayed but no exception is raised. If they are both False, then function is silently skipped.

If raise_ex is True, the value of *warn* is not taken into account.

Typical usage

The user should decorate functions or class methods when they cannot be executed without the optional library. In the specific case of Processor subclass, where the class itself can be created also without the missing library, but it is required somewhere in the processor execution, then the user is suggested to decorate the execute method with this decorator.

Parameters

- **module_name** (*str*) – The optional module(s) from which the function depends on. A ";" separated list of modules can also be provided.
- **raise_ex** (*bool, Optional*) – Flag to raise an exception if module_name is not found, defaults to False.

- **warn** (*bool, Optional*) – Flag to display a warning message if module_name is not found, default to True.

Returns
The wrapped function

Return type
Callable

Raises
ImportError – if module_name is not found and raise_ex is True.

mafw.decorators.**execution_workflow**(*loop_type:* LoopType | *str = LoopType.ForLoop*)

A decorator factory for the definition of the looping strategy.

This decorator factory must be applied to Processor subclasses to modify their value of loop_type in order to change the execution workflow.

See *single_loop()*, *for_loop()* and *while_loop()* decorator shortcuts.

Parameters
loop_type (*LoopType | str, Optional*) – The type of execution workflow requested for the decorated class. Defaults to LoopType.ForLoop.

mafw.decorators.**for_loop**(*cls*)

A decorator shortcut to define a for loop execution processor.

mafw.decorators.**orphan_protector**(*cls*)

A class decorator to modify the init method of a Processor so that the remove_orphan_files is set to False and no orphan files will be removed.

mafw.decorators.**parallel_for_loop**(*cls*)

A decorator shortcut to define a parallel for loop execution processor.

mafw.decorators.**parallel_for_loop_with_queue**(*cls*)

A decorator shortcut to define a parallel for loop with queue execution processor.

mafw.decorators.**processor_depends_on_optional**(*module_name: str*, *raise_ex: bool = False*, *warn: bool = True*) → Callable[[type[*P*]], type[*Processor*]]

Class decorator factory to check if module module_name is available.

It checks if all the optional modules listed in *module_name* separated by a ';' can be found.

If all modules are found, then the class is returned as it is.

If at least one module is not found:

- and raise_ex is True, an ImportError exception is raised and the user is responsible to deal with it.
- if raise_ex is False, instead of returning the class, the *Processor* is returned.
- depending on the value of warn, the user will be informed with a warning message or not.

Typical usage

The user should decorate Processor subclasses everytime the optional module is required in their __init__ method. Should the check on the optional module have a positive outcome, then the Processor subclass is returned. Otherwise, if raise_ex is False, an instance of the base *Processor* is returned. In this way, the returned class can still be executed without breaking the execution scheme but of course, without producing any output.

Should be possible to run the __init__ method of the class without the missing library, then the user can also follow the approach described in this other *example*.

Parameters

- **module_name** (*str*) – The optional module(s) from which the class depends on. A ";" separated list of modules can also be provided.
- **raise_ex** (*bool, Optional*) – Flag to raise an exception if module_name not found, defaults to False.
- **warn** (*bool, Optional*) – Flag to display a warning message if module_name is not found, defaults to True.

Returns
The wrapped processor.

Return type
type(*processor.Processor*)

Raises
ImportError – if module_name is not found and raise_ex is True.

mafw.decorators.**single_loop**(*cls*)

A decorator shortcut to define a single execution processor.

mafw.decorators.**singleton**(*cls*)

Make a class a Singleton class (only one instance)

mafw.decorators.**suppress_warnings**(*func: F*) → *F*

Decorator to suppress warnings during the execution of a test function.

This decorator uses the *warnings.catch_warnings()* context manager to temporarily change the warning filter to ignore all warnings. It is useful when you want to run a test without having warnings clutter the output.

Usage:

```
@suppress_warnings
def test_function():
    # Your test code that might emit warnings
```

Parameters
func (*Callable*) – The test function to be decorated.

Returns
The wrapped function with suppressed warnings.

Return type
Callable

mafw.decorators.**while_loop**(*cls*)

A decorator shortcut to define a while loop execution processor.

14.1.4 mafw.devtools

MAFw development tools package.

This package contains the business logic modules used by the `release-mgt` and `multiversion-doc` CLI scripts. It requires optional development dependencies (`requests`, `packaging`) that are not part of the core MAFw runtime.

Use *ensure_devtools_available()* as a guard before importing subpackages.

Functions

ensure_devtools_available()	Verify that optional development dependencies are importable.

Exceptions

DevtoolsError	Base exception for MAFw development tool errors.

exception mafw.devtools.**DevtoolsError**

Bases: Exception

Base exception for MAFw development tool errors.

This exception replaces direct usage of `click.ClickException` in business-logic modules so that the library layer remains independent of the CLI framework.

mafw.devtools.**_read_devtools_deps**() → list[str]

Read the `devtools` optional-dependency names from MAFw package metadata.

This function requires `packaging` to be importable (the caller ensures this). It uses `Requirement` to parse the dependency specifiers reliably.

Falls back to reading `pyproject.toml` from the source tree if the installed metadata does not yet include the `devtools` extra (e.g. stale editable install).

Returns
List of top-level package names required by the `devtools` extra.

Return type
list[str]

mafw.devtools.**ensure_devtools_available**() → None

Verify that optional development dependencies are importable.

The list of required packages is read from MAFw's own `[devtools]` optional-dependency group in `pyproject.toml`, keeping it as the single source of truth. The check is performed only once; subsequent calls return immediately.

Raises
ImportError – If `packaging` or any other package from the `devtools` extra cannot be imported.

mafw.devtools.**_devtools_checked: bool = True**

Module-level flag to skip repeated import checks after the first successful call.

Modules

cli	CLI sub-package for MAFw development tools.
dependencies	Dependency management utilities for MAFw development tools.
documentation	Documentation management utilities for MAFw development tools.
git	Git operations for MAFw development tools.
gitlab	GitLab integration helpers for MAFw development tools.
release	Release management utilities for MAFw development tools.
toolchain	Toolchain management package for MAFw development tools.

mafw.devtools.cli

CLI sub-package for MAFw development tools.

Modules

cli	Unified development tools CLI for MAFw maintenance tasks.
completion	Shell completion management for the `devtools` CLI.
dependencies	Dependency verification and maintenance CLI commands.
documentation	CLI sub-package for building and managing versioned documentation.
release	CLI sub-package for release lifecycle management.
toolchain	Toolchain management CLI commands.

mafw.devtools.cli.cli

Unified development tools CLI for MAFw maintenance tasks.

mafw.devtools.cli.completion

Shell completion management for the `devtools` CLI.

mafw.devtools.cli.dependencies

Dependency verification and maintenance CLI commands.

Module Attributes

CONTEXT_SETTINGS	Click context settings for command line help aliases.

mafw.devtools.cli.dependencies.**CONTEXT_SETTINGS = {'help_option_names': ['-h', '--help']}**

Click context settings for command line help aliases.

Modules

audit	Audit dependency vulnerabilities using pip-audit.
freeze	Freeze dependency upper bounds in the project TOML.
latest	Group for latest-dependency verification commands.
oldest	Group for oldest-dependency commands.
registry	GitLab registry command group for dependency reference files.

mafw.devtools.cli.dependencies.audit

mafw.devtools.cli.dependencies.**audit = <Command audit>**

Audit dependency vulnerabilities using pip-audit.

mafw.devtools.cli.dependencies.freeze

mafw.devtools.cli.dependencies.**freeze = <Command freeze>**

Freeze dependency upper bounds in the project TOML.

mafw.devtools.cli.dependencies.latest

mafw.devtools.cli.dependencies.**latest = <AbbreviateGroup latest>**

Group for latest-dependency verification commands.

mafw.devtools.cli.dependencies.oldest

mafw.devtools.cli.dependencies.**oldest = <AbbreviateGroup oldest>**

Group for oldest-dependency commands.

mafw.devtools.cli.dependencies.registry

mafw.devtools.cli.dependencies.**registry = <AbbreviateGroup registry>**

GitLab registry command group for dependency reference files.

mafw.devtools.cli.documentation

CLI sub-package for building and managing versioned documentation.

Module Attributes

CONTEXT_SETTINGS	Click context settings for command line help aliases.

mafw.devtools.cli.documentation.**CONTEXT_SETTINGS = {'help_option_names': ['-h', '--help']}**

Click context settings for command line help aliases.

Modules

build	Build multiversion documentation.
prune	Prune old documentation versions to stay within size limit.
registry	Interact with the GitLab Generic Package Registry for mafw-docs packages.
server	Manage a local web server for browsing documentation over HTTP.

mafw.devtools.cli.documentation.build

mafw.devtools.cli.documentation.**build = <Command build>**

Build multiversion documentation.

mafw.devtools.cli.documentation.prune

mafw.devtools.cli.documentation.**prune = <Command prune>**

Prune old documentation versions to stay within size limit.

mafw.devtools.cli.documentation.registry

mafw.devtools.cli.documentation.**registry = <AbbreviateGroup registry>**

Interact with the GitLab Generic Package Registry for mafw-docs packages.

mafw.devtools.cli.documentation.server

mafw.devtools.cli.documentation.**server = <AbbreviateGroup server>**

Manage a local web server for browsing documentation over HTTP.

mafw.devtools.cli.release

CLI sub-package for release lifecycle management.

Module Attributes

CONTEXT_SETTINGS	Click context settings for command line help aliases.

mafw.devtools.cli.release.**CONTEXT_SETTINGS = {'help_option_names': ['-h', '--help']}**

Click context settings for command line help aliases.

Modules

create	Prepare a new MAFw release from the main branch.

mafw.devtools.cli.release.create

mafw.devtools.cli.release.**create = <Command create>**

Prepare a new MAFw release from the main branch.

mafw.devtools.cli.toolchain

Toolchain management CLI commands.

This package provides the `devtools toolchain` Click command group and its subcommands (`check`, `update`, `verify`, `bootstrap`, `list`). The group-level `--include-host` / `--exclude-host` flags control whether host-category tools participate in processing, while the *common_options()* decorator provides repeatable `-i` / `--include` and `-e` / `--exclude` options for per-tool name filtering.

The default *ToolRegistry* is populated via *get_default_registry()* and stored in the Click context for access by subcommands.

mafw.devtools.cli.toolchain.**common_options**(*fn: Any*) → Any

Shared decorator applying `-i`/`--include` and `-e`/`--exclude` options.

These repeatable options allow subcommands to filter which tools are processed by name. They are mutually exclusive: specifying both in the same invocation triggers a `UsageError` at filtering time (enforced by *resolve_tools()*).

The decorated function receives `include` and `exclude` keyword arguments as tuples of tool-name strings.

Modules

bootstrap_cmd	Bootstrap subcommand for the toolchain command group.

continues on next page

Table 14.39 – continued from previous page

check_cmd	Check subcommand for the toolchain command group.
list_cmd	The `list` subcommand for the toolchain CLI group.
update_cmd	The `update` subcommand for the toolchain CLI group.
verify_cmd	Verify subcommand for the toolchain command group.

mafw.devtools.cli.toolchain.bootstrap_cmd

Bootstrap subcommand for the toolchain command group.

This module implements the `devtools toolchain bootstrap` command, which iterates over all applicable *ToolChainTool* instances and attempts to install any that are not yet present. Each tool's *bootstrap()* method is called; the result is categorised as successfully installed, already present, or failed.

A Rich spinner is displayed while each tool is being processed, providing visual feedback during potentially slow installation operations.

Project tools raise *DevtoolsError* from `bootstrap()` to indicate they do not support standalone bootstrapping (they are managed through pyproject.toml and installed automatically via Hatch environments). This is treated as a skip and does not affect the exit code.

mafw.devtools.cli.toolchain.check_cmd

Check subcommand for the toolchain command group.

This module implements the `devtools toolchain check` CLI command, which iterates over applicable tools, detects their current and latest versions, and displays a Rich table summarising each tool's synchronisation status.

A Rich progress spinner provides visual feedback during version detection (which may involve PyPI queries or subprocess calls). The command exits with code 0 if all tools are in sync, or code 1 if any tool is out of sync or encountered a detection error.

mafw.devtools.cli.toolchain.list_cmd

The `list` subcommand for the toolchain CLI group.

This module implements the `devtools toolchain list` command, which displays a Rich table of all applicable tools with their name and category. Tools are filtered by the group-level `--include-host` / `--exclude-host` flags and the subcommand-level `-i` / `-e` options, then displayed in their registered order.

mafw.devtools.cli.toolchain.update_cmd

The `update` subcommand for the toolchain CLI group.

This module implements the `devtools toolchain update` command, which iterates over all applicable *ToolChainTool* instances and attempts to bring each tool's configuration to its latest version. For each tool that reports a change, the optional *post_update()* hook is executed. Failures are accumulated and reported via Rich console output, with a non-zero exit code when any tool encountered an error.

A Rich progress spinner provides visual feedback during processing.

mafw.devtools.cli.toolchain.verify_cmd

Verify subcommand for the toolchain command group.

This module implements the `devtools toolchain verify` command, which iterates over all applicable tools with a Rich progress spinner, calls each tool's *verify()* method, and reports any configuration inconsistencies found. Exceptions during verification are treated as implicit issues. The exit code reflects whether any issues were detected (exit 1) or all tools are consistent (exit 0).

mafw.devtools.dependencies

Dependency management utilities for MAFw development tools.

This package provides business logic for compiling, freezing, unfreezing, comparing, and auditing project dependencies.

class mafw.devtools.dependencies.**PackageChange**(*package_name: str*, *change_type: Literal['ADDED', 'REMOVED', 'UPDATED']*, *reference_version: str | None*, *new_version: str | None*)

Bases: `object`

A single dependency change between reference and latest lockfiles.

Each instance represents one package that was added, removed, or updated between the reference and the latest resolved dependency set.

Parameters

- **package_name** – Normalized (lowercase) package name.
- **change_type** – Classification of the change (ADDED, REMOVED, UPDATED).
- **reference_version** – Version in the reference file, `None` for ADDED entries.
- **new_version** – Version in the latest file, `None` for REMOVED entries.

class mafw.devtools.dependencies.**VersionComparisonResult**(*python_version: str*, *changes: list[~mafw.devtools.dependencies.compare.PackageChan = <factory>*)

Bases: `object`

Comparison result for a single Python version.

Holds all detected dependency changes for a given Python interpreter version.

Parameters

- **python_version** – Python version string (e.g. `"3.12"`).
- **changes** – Sorted list of package changes for this version.

property has_changes: bool

Return `True` if there is at least one dependency change.

mafw.devtools.dependencies.**collect_compiled_dependency_versions**(*python_versions: list[str]*) → dict[str, Version]

Compile dependency lockfiles and collect the resolved versions they report.

The command mirrors the dependency verification workflow by invoking `uv pip compile` for each supported CPython version. A temporary lockfile is generated for each version and removed afterwards.

Parameters
python_versions (*list[str]*) – Supported Python versions to compile.

Returns
Mapping of package name to highest resolved version across the compiled lockfiles.

Return type
dict[str, Version]

mafw.devtools.dependencies.**commit_dependency_unfreeze**(*dry_run: bool*) → None

Commit the unfreezing of dependency upper bounds to `main`.

Parameters
dry_run (*bool*) – Whether command execution is disabled.

mafw.devtools.dependencies.**compare_packages**(*latest_packages: dict[str, dict[str, Any]]*, *reference_packages: dict[str, dict[str, Any]]*) → list[*PackageChange*]

Compare two package dictionaries and classify differences.

Both dictionaries are expected to be keyed by **lowercase** package name. The function classifies each difference into one of three categories:

- **ADDED**: package present in *latest* but absent from *reference*.
- **REMOVED**: package present in *reference* but absent from *latest*.
- **UPDATED**: package present in both but with a different version or marker.

Parameters

- **latest_packages** (*dict[str, dict[str, Any]]*) – Packages from the freshly compiled lockfile, keyed by lowercase name.
- **reference_packages** (*dict[str, dict[str, Any]]*) – Packages from the reference lockfile, keyed by lowercase name.

Returns
List of *PackageChange* entries sorted alphabetically by package name.

Return type
list[*PackageChange*]

mafw.devtools.dependencies.**compile_dependency_lockfile**(*python_version: str*, *pylock_file: Path*, *extras: list[str]*, *resolution: str | None = None*, *output_format: str | None = None*, *with_hashes: bool = False*) → None

Compile a dependency lockfile for a specific CPython version.

Parameters

- **python_version** (*str*) – CPython version used for the `uv pip compile` run.
- **pylock_file** (*Path*) – Output path for the generated lockfile.
- **extras** (*list[str]*) – Project extras requested during compilation.
- **resolution** (*str | None*) – Optional UV resolution strategy (e.g. 'lowest-direct', 'highest').
- **output_format** (*str | None*) – Optional output format (e.g. 'requirements.txt').
- **with_hashes** (*bool*) – Whether to generate hashes for the compiled requirements.

mafw.devtools.dependencies.**compile_python_selector**(*python_version: str*) → str

Build the Python selector used by uv for dependency compilation.

For Python 3.14 and newer, request the GIL-enabled variant explicitly so free-threaded interpreters do not leak into dependency resolution.

This distinction is still needed because of the psycopg

Parameters
python_version (*str*) – Base Python version in `major.minor` form.

Returns
Python selector string passed to uv.

Return type
str

mafw.devtools.dependencies.**compute_upper_bound**(*lower_bound: str*) → str

Compute an upper bound for a dependency based on PEP 440 compatible-release philosophy.

The rule implemented here is purposely conservative and mirrors the intent of compatible release clauses while remaining explicit:

- For major-versioned releases (`X.*` with `X > 0`), freeze to `<(X + 1)`.
- For `0.*` releases, freeze to `<0.(minor + 1)` (rolling compatibility during pre-1.0).

Parameters
lower_bound (*str*) – Version string used as the starting point for the freeze rule.

Returns
Upper-bound version string without operator.

Return type
str

Raises
DevtoolsError – If the version cannot be parsed.

mafw.devtools.dependencies.**ensure_mafw_project_root**() → list[str]

Ensure the current working directory is the MAFw project root.

Returns
Validated supported Python versions from `tool.mafw.supported-python`.

Return type
list[str]

Raises
DevtoolsError – If `pyproject.toml` is missing or does not identify MAFw.

mafw.devtools.dependencies.**freeze_dependencies**(*, *dry_run: bool*) → str

Freeze dependencies by adding computed upper bounds in `pyproject.toml`.

Parameters
dry_run (*bool*) – Whether command execution is disabled.

Returns
Original `pyproject.toml` content captured before freeze.

Return type
str

mafw.devtools.dependencies.**freeze_pyproject_toml**(*toml_text: str*, *, *resolved_versions: dict[str, Version] | None = None*, *doc: TOMLDocument | None = None*) → tuple[str, list[str]]

Freeze dependencies in a `pyproject.toml` payload by adding upper bounds.

The TOML structure is preserved via tomlkit; only dependency strings may be normalized.

When `resolved_versions` is provided, the function uses those compiled versions as the basis for upper-bound computation. Otherwise the declared lower bounds are used as a fallback.

Parameters

- **toml_text** (*str*) – Raw TOML file content.
- **resolved_versions** (*dict[str, Version] | None*) – Optional mapping of dependency names to resolved versions.
- **doc** (*tomlkit.TOMLDocument | None*) – Optional pre-parsed TOML document to reuse when available.

Returns
Updated TOML plus warnings.

Return type
tuple[str, list[str]]

Raises
DevtoolsError – If TOML parsing fails.

mafw.devtools.dependencies.**freeze_requirement**(*requirement_text: str*, *, *resolved_version: str | None = None*) → tuple[str, list[str]]

Add a computed upper bound to a requirement string, if eligible.

The function is intentionally conservative:

- URL-based requirements are skipped (cannot be version constrained).
- Requirements already containing <, <=, ~=, == or === are skipped.
- Requirements without a lower bound emit a warning and are left unchanged.
- When a resolved version is provided, the upper bound is computed from that version instead of the declared lower bound.

Parameters
requirement_text (*str*) – Raw PEP 508 requirement string.

Returns
Updated requirement text plus warnings.

Return type
tuple[str, list[str]]

Raises
DevtoolsError – If parsing fails.

mafw.devtools.dependencies.**get_expected_lower_bounds**() → dict[str, Requirement]

Read pyproject.toml and extract dependencies with their expected lower bounds.

Returns
A dictionary mapping lowercase package names to their parsed Requirement objects.

Return type
dict[str, Requirement]

Raises
DevtoolsError – If the pyproject.toml cannot be parsed.

mafw.devtools.dependencies.**load_pylock_packages**(*pylock_path: Path*) → dict[str, dict[str, Any]]

Parse a pylock TOML file into a dictionary keyed by lowercase package name.

Each value in the returned dictionary contains at minimum the `name`, `version`, and optionally `marker` fields from the original TOML entry.

Parameters
pylock_path (*Path*) – Path to the pylock TOML file.

Returns
Dictionary mapping lowercase package names to their package metadata.

Return type
dict[str, dict[str, Any]]

Raises

- **FileNotFoundError** – If *pylock_path* does not exist.
- **tomllib.TOMLDecodeError** – If the file is not valid TOML.

mafw.devtools.dependencies.**load_pyproject_doc**(*toml_text: str*) → TOMLDocument

Parse a `pyproject.toml` payload once and return the TOML document.

Parameters
toml_text (*str*) – Raw TOML payload.

Returns
Parsed TOML document.

Return type
tomlkit.TOMLDocument

Raises
DevtoolsError – If the TOML payload cannot be parsed.

mafw.devtools.dependencies.**parse_python_version**(*version: str*) → tuple[int, int]

Parse a Python version string in `major.minor` form.

Parameters
version (*str*) – Python version string.

Returns
Major/minor version tuple.

Return type
tuple[int, int]

Raises
DevtoolsError – If the version is invalid.

mafw.devtools.dependencies.**project_python_versions**() → list[str]

Read the supported CPython versions from `tool.mafw.supported-python` list in pyproject.toml.

The field is expected to be a list of strings representing CPython major.minor versions. The helper validates each entry and returns a sorted, de-duplicated list so downstream callers have deterministic ordering.

Returns
Sorted list of supported CPython versions.

Return type
list[str]

Raises
DevtoolsError – If `pyproject.toml` cannot be parsed or the field contains unsupported values.

mafw.devtools.dependencies.**project_python_versions_from_doc**(*doc: TOMLDocument*) → list[str]

Extract supported CPython versions from a parsed `pyproject.toml` document.

Parameters
doc (*tomlkit.TOMLDocument*) – Parsed TOML document.

Returns
Sorted list of supported CPython versions.

Return type
list[str]

Raises
DevtoolsError – If the `tool.mafw.supported-python` field is invalid.

mafw.devtools.dependencies.**python_versions_between**(*min_python_ver: str*, *max_python_ver: str*, *supported_versions: list[str]*) → list[str]

Build the inclusive list of Python versions between two bounds.

The returned versions must also be present in `supported_versions`.

Parameters

- **min_python_ver** (*str*) – Minimum Python version.
- **max_python_ver** (*str*) – Maximum Python version.
- **supported_versions** (*list[str]*) – List of supported Python versions from project metadata.

Returns
Ordered list of version strings.

Return type
list[str]

mafw.devtools.dependencies.**read_compiled_dependency_versions**(*pylock_text: str*) → dict[str, Version]

Read resolved dependency versions from a compiled lockfile payload.

The returned mapping stores the highest resolved version seen for each dependency name, normalized to lowercase for stable lookups.

Parameters
pylock_text (*str*) – Raw `pylock.py3_14.toml` content.

Returns
Mapping of package name to highest resolved version.

Return type
dict[str, Version]

Raises
DevtoolsError – If the TOML payload cannot be parsed.

mafw.devtools.dependencies.**render_comparison_json**(*results: list[VersionComparisonResult]*) → str

Render comparison results as a JSON document string.

Produces a JSON object with:

- `timestamp`: ISO 8601 generation time.
- `python_versions`: ordered list of all Python versions tested.
- `results`: dictionary keyed by Python version, each value being a list of change entries (empty list when no differences exist for that version).

Each change entry contains `package_name`, `change_type`, `reference_version` (null for ADDED), and `new_version` (null for REMOVED).

Parameters
results (*list[VersionComparisonResult]*) – Comparison results for each Python version.

Returns
Formatted JSON string with 2-space indentation.

Return type
str

mafw.devtools.dependencies.**render_comparison_markdown**(*results: list[VersionComparisonResult]*) → str

Render comparison results as a markdown document string.

Produces a structured markdown report with a title, an ISO 8601 timestamp, and per-version sections containing ADDED, REMOVED, and UPDATED tables. Versions with no dependency changes are omitted from the output.

Parameters
results (*list[VersionComparisonResult]*) – List of comparison results, one per Python version.

Returns
Complete markdown document as a string.

Return type
str

mafw.devtools.dependencies.**render_comparison_rich**(*results: list[VersionComparisonResult]*, *console: Console*) → None

Render comparison results to the terminal using Rich panels and tables.

Displays a title panel, followed by a section per Python version containing ADDED, REMOVED, and UPDATED subsections in that fixed order. Within each subsection, entries are sorted alphabetically by package name.

When no changes are detected across all Python versions, a single informational message is printed instead.

Parameters

- **results** (*list[VersionComparisonResult]*) – Comparison results for each Python version.
- **console** (*Console*) – Rich Console instance for output.

mafw.devtools.dependencies.**resolve_output_format**(*explicit_format: str | None*, *output_file: Path | None*) → str

Determine the effective output format based on CLI arguments.

The resolution follows a priority order that avoids ambiguity between an explicitly requested format and the file extension of the output path:

1. If *output_file* has a recognized extension **and** *explicit_format* is set:
   - **Match** (e.g., `json` + `.json`): return the format.
   - **Conflict** (e.g., `json` + `.md`): raise `click.UsageError`.
2. If *output_file* has a recognized extension **and** *explicit_format* is `None`: infer the format from the extension.
3. If *output_file* has an unrecognized extension **and** *explicit_format* is set: return the explicit format.
4. Otherwise: return `"markdown"` as the default.

Parameters

- **explicit_format** (*str | None*) – The value of `--format` if explicitly provided by the user, or `None` when omitted.
- **output_file** (*Path | None*) – The value of `--output-file`, or `None` when omitted.

Returns
The resolved format string (`"markdown"` or `"json"`).

Return type
str

Raises
click.UsageError – When *explicit_format* conflicts with the format inferred from the file extension.

mafw.devtools.dependencies.**run_pip_audit**(*req_file: Path*, *output_file: Path*, *output_format: str*) → CompletedProcess[Any]

Run pip-audit on a requirements file and save the output.

Parameters

- **req_file** (*Path*) – Requirements file to audit.
- **output_file** (*Path*) – Path to the output report.

- **output_format** (*str*) – Format of the report (e.g. 'markdown', 'json').

Returns
Completed process produced by the command execution.

Return type
subprocess.CompletedProcess[Any]

mafw.devtools.dependencies.**unfreeze_dependencies**(*original_pyproject_toml: str*, *, *dry_run: bool*) → None

Unfreeze dependencies by removing computed upper bounds in `pyproject.toml`.

Parameters
dry_run (*bool*) – Whether command execution is disabled.

mafw.devtools.dependencies.**unfreeze_pyproject_toml**(*toml_text: str*, *, *baseline_toml_text: str | None = None*, *doc: TOMLDocument | None = None*, *baseline_doc: TOMLDocument | None = None*) → tuple[str, list[str]]

Unfreeze dependencies in a `pyproject.toml` payload by removing computed upper bounds.

Only upper bounds matching the computed rule are removed; existing manual constraints remain.

Parameters

- **toml_text** (*str*) – Raw TOML file content.
- **baseline_toml_text** (*str | None*) – Optional original TOML text captured before freezing. When provided, unfreezing is computed against the baseline to avoid altering dependencies that were already frozen before release.
- **doc** (*tomlkit.TOMLDocument | None*) – Optional pre-parsed TOML document to reuse when available.
- **baseline_doc** (*tomlkit.TOMLDocument | None*) – Optional pre-parsed baseline TOML document to reuse when available.

Returns
Updated TOML plus warnings.

Return type
tuple[str, list[str]]

Raises
DevtoolsError – If TOML parsing fails.

mafw.devtools.dependencies.**unfreeze_requirement**(*requirement_text: str*) → tuple[str, list[str]]

Remove a computed upper bound from a requirement string, if it matches the computed rule.

Only the auto-generated upper bound <`upper` is removed; existing manual upper bounds are preserved.

Parameters
requirement_text (*str*) – Raw PEP 508 requirement string.

Returns
Updated requirement text plus warnings.

Return type
tuple[str, list[str]]

Raises
DevtoolsError – If parsing fails.

mafw.devtools.dependencies.**update_requirements_and_readme**(*, *dry_run: bool*) → None

Update the requirements RST files and README.rst.

Parameters
dry_run (*bool*) – Whether command execution is disabled.

mafw.devtools.dependencies.**verify_lowest_resolution**(*env_name: str*, *python_version: str*, *expected_bounds: dict[str, Requirement]*) → None

Verify that the specified environment has the expected lowest dependency versions.

This function executes *uv pip list* in the given environment, parses the output, and compares installed versions against the expected lower bounds extracted from *pyproject.toml*.

Parameters

- **env_name** (*str*) – The name of the environment (e.g., 'hatch-test', 'types').
- **python_version** (*str*) – The Python version string (e.g., '3.11').
- **expected_bounds** (*dict[str, Requirement]*) – Mapping of package names to their Requirement objects.

Raises

DevtoolsError – If a required dependency is missing.

Modules

audit	Dependency auditing utilities for MAFw.
compare	Dependency comparison engine for MAFw lockfiles.
compile	Dependency compilation utilities for MAFw.
freeze	Dependency freezing and unfreezing utilities for MAFw.
verify	Dependency resolution verification utilities for MAFw.

mafw.devtools.dependencies.audit

Dependency auditing utilities for MAFw.

This module provides functions for running pip-audit against compiled requirement files.

Functions

run_pip_audit(req_file, output_file, ...)	Run pip-audit on a requirements file and save the output.

mafw.devtools.dependencies.audit.**run_pip_audit**(*req_file: Path*, *output_file: Path*, *output_format: str*) → CompletedProcess[Any]

Run pip-audit on a requirements file and save the output.

Parameters

- **req_file** (*Path*) – Requirements file to audit.
- **output_file** (*Path*) – Path to the output report.
- **output_format** (*str*) – Format of the report (e.g. 'markdown', 'json').

Returns

Completed process produced by the command execution.

Return type

subprocess.CompletedProcess[Any]

mafw.devtools.dependencies.compare

Dependency comparison engine for MAFw lockfiles.

This module provides data models and rendering functions for comparing dependency lockfiles across Python versions. It supports JSON, markdown, and Rich terminal output formats.

Module Attributes

EXTENSION_FORMAT_MAP	Mapping from recognized file extensions to output format identifiers.

Functions

compare_packages(latest_packages, ...)	Compare two package dictionaries and classify differences.
render_comparison_json(results)	Render comparison results as a JSON document string.
render_comparison_markdown(results)	Render comparison results as a markdown document string.
render_comparison_rich(results, console)	Render comparison results to the terminal using Rich panels and tables.
resolve_output_format(explicit_format, ...)	Determine the effective output format based on CLI arguments.

Classes

PackageChange(package_name, change_type, ...)	A single dependency change between reference and latest lockfiles.
VersionComparisonResult(python_version, changes)	Comparison result for a single Python version.

class mafw.devtools.dependencies.compare.**PackageChange**(*package_name: str*, *change_type: Literal['ADDED', 'REMOVED', 'UPDATED']*, *reference_version: str | None*, *new_version: str | None*)

Bases: `object`

A single dependency change between reference and latest lockfiles.

Each instance represents one package that was added, removed, or updated between the reference and the latest resolved dependency set.

Parameters

- **package_name** – Normalized (lowercase) package name.
- **change_type** – Classification of the change (ADDED, REMOVED, UPDATED).
- **reference_version** – Version in the reference file, `None` for ADDED entries.
- **new_version** – Version in the latest file, `None` for REMOVED entries.

class mafw.devtools.dependencies.compare.**VersionComparisonResult**(*python_version: str*, *changes: list[~mafw.devtools.dependencies.compare.Pa = <factory>*)

Bases: `object`

Comparison result for a single Python version.

Holds all detected dependency changes for a given Python interpreter version.

Parameters

- **python_version** – Python version string (e.g. `"3.12"`).
- **changes** – Sorted list of package changes for this version.

property has_changes: bool

Return `True` if there is at least one dependency change.

mafw.devtools.dependencies.compare.**compare_packages**(*latest_packages: dict[str, dict[str, Any]]*, *reference_packages: dict[str, dict[str, Any]]*) → list[*PackageChange*]

Compare two package dictionaries and classify differences.

Both dictionaries are expected to be keyed by **lowercase** package name. The function classifies each difference into one of three categories:

- **ADDED**: package present in *latest* but absent from *reference*.
- **REMOVED**: package present in *reference* but absent from *latest*.
- **UPDATED**: package present in both but with a different version or marker.

Parameters

- **latest_packages** (*dict[str, dict[str, Any]]*) – Packages from the freshly compiled lockfile, keyed by lowercase name.
- **reference_packages** (*dict[str, dict[str, Any]]*) – Packages from the reference lockfile, keyed by lowercase name.

Returns
List of *PackageChange* entries sorted alphabetically by package name.

Return type
list[*PackageChange*]

mafw.devtools.dependencies.compare.**render_comparison_json**(*results: list[VersionComparisonResult]*) → str

Render comparison results as a JSON document string.

Produces a JSON object with:

- `timestamp`: ISO 8601 generation time.
- `python_versions`: ordered list of all Python versions tested.
- `results`: dictionary keyed by Python version, each value being a list of change entries (empty list when no differences exist for that version).

Each change entry contains `package_name`, `change_type`, `reference_version` (null for ADDED), and `new_version` (null for REMOVED).

Parameters
results (*list[VersionComparisonResult]*) – Comparison results for each Python version.

Returns
Formatted JSON string with 2-space indentation.

Return type
str

mafw.devtools.dependencies.compare.**render_comparison_markdown**(*results: list[VersionComparisonResult]*) → str

Render comparison results as a markdown document string.

Produces a structured markdown report with a title, an ISO 8601 timestamp, and per-version sections containing ADDED, REMOVED, and UPDATED tables. Versions with no dependency changes are omitted from the output.

Parameters
results (*list[VersionComparisonResult]*) – List of comparison results, one per Python version.

Returns
Complete markdown document as a string.

Return type
str

mafw.devtools.dependencies.compare.**render_comparison_rich**(*results: list[VersionComparisonResult]*, *console: Console*) → None

Render comparison results to the terminal using Rich panels and tables.

Displays a title panel, followed by a section per Python version containing ADDED, REMOVED, and UPDATED subsections in that fixed order. Within each subsection, entries are sorted alphabetically by package name.

When no changes are detected across all Python versions, a single informational message is printed instead.

Parameters

- **results** (*list[VersionComparisonResult]*) – Comparison results for each Python version.
- **console** (*Console*) – Rich Console instance for output.

mafw.devtools.dependencies.compare.**resolve_output_format**(*explicit_format: str | None*, *output_file: Path | None*) → str

Determine the effective output format based on CLI arguments.

The resolution follows a priority order that avoids ambiguity between an explicitly requested format and the file extension of the output path:

1. If *output_file* has a recognized extension **and** *explicit_format* is set:
   - **Match** (e.g., `json` + `.json`): return the format.
   - **Conflict** (e.g., `json` + `.md`): raise `click.UsageError`.
2. If *output_file* has a recognized extension **and** *explicit_format* is `None`: infer the format from the extension.
3. If *output_file* has an unrecognized extension **and** *explicit_format* is set: return the explicit format.
4. Otherwise: return `"markdown"` as the default.

Parameters

- **explicit_format** (*str | None*) – The value of `--format` if explicitly provided by the user, or `None` when omitted.
- **output_file** (*Path | None*) – The value of `--output-file`, or `None` when omitted.

Returns
The resolved format string (`"markdown"` or `"json"`).

Return type
str

Raises
click.UsageError – When *explicit_format* conflicts with the format inferred from the file extension.

mafw.devtools.dependencies.compare.**EXTENSION_FORMAT_MAP: dict[str, str] = {'.json': 'json', '.md': 'markdown'}**

Mapping from recognized file extensions to output format identifiers.

mafw.devtools.dependencies.compile

Dependency compilation utilities for MAFw.

This module provides functions for compiling dependency lockfiles using uv, reading resolved dependency versions, and managing Python version metadata from the project configuration.

Module Attributes

PYPROJECT_FILE	Path to the TOML file containing the project dependencies.
DEFAULT_FREEZE_EXTRAS	Extras used when compiling dependency lockfiles for release freezing and compatibility checks.

Functions

collect_compiled_dependency_versions(...)	Compile dependency lockfiles and collect the resolved versions they report.
compile_dependency_lockfile(python_version, ...)	Compile a dependency lockfile for a specific CPython version.
compile_python_selector(python_version)	Build the Python selector used by uv for dependency compilation.
ensure_mafw_project_root()	Ensure the current working directory is the MAFw project root.
load_pylock_packages(pylock_path)	Parse a pylock TOML file into a dictionary keyed by lowercase package name.
load_pyproject_doc(toml_text)	Parse a `pyproject.toml` payload once and return the TOML document.
parse_python_version(version)	Parse a Python version string in `major.minor` form.
project_python_versions()	Read the supported CPython versions from `tool.mafw.supported-python` list in pyproject.toml.
project_python_versions_from_doc(doc)	Extract supported CPython versions from a parsed `pyproject.toml` document.
python_versions_between(min_python_ver, ...)	Build the inclusive list of Python versions between two bounds.
read_compiled_dependency_versions(pylock_tex	Read resolved dependency versions from a compiled lockfile payload.

mafw.devtools.dependencies.compile.**collect_compiled_dependency_versions**(*python_versions: list[str]*) → dict[str, Version]

Compile dependency lockfiles and collect the resolved versions they report.

The command mirrors the dependency verification workflow by invoking `uv pip compile` for each supported CPython version. A temporary lockfile is generated for each version and removed afterwards.

Parameters
python_versions (*list[str]*) – Supported Python versions to compile.

Returns
Mapping of package name to highest resolved version across the compiled lockfiles.

Return type
dict[str, Version]

mafw.devtools.dependencies.compile.**compile_dependency_lockfile**(*python_version: str*, *pylock_file: Path*, *extras: list[str]*, *resolution: str | None = None*, *output_format: str | None = None*, *with_hashes: bool = False*) → None

Compile a dependency lockfile for a specific CPython version.

Parameters

- **python_version** (*str*) – CPython version used for the `uv pip compile` run.
- **pylock_file** (*Path*) – Output path for the generated lockfile.
- **extras** (*list[str]*) – Project extras requested during compilation.
- **resolution** (*str | None*) – Optional UV resolution strategy (e.g. 'lowest-direct', 'highest').
- **output_format** (*str | None*) – Optional output format (e.g. 'requirements.txt').
- **with_hashes** (*bool*) – Whether to generate hashes for the compiled requirements.

mafw.devtools.dependencies.compile.**compile_python_selector**(*python_version: str*) → str

Build the Python selector used by uv for dependency compilation.

For Python 3.14 and newer, request the GIL-enabled variant explicitly so free-threaded interpreters do not leak into dependency resolution.

This distinction is still needed because of the psycopg

Parameters
python_version (*str*) – Base Python version in `major.minor` form.

Returns
Python selector string passed to uv.

Return type
str

mafw.devtools.dependencies.compile.**ensure_mafw_project_root**() → list[str]

Ensure the current working directory is the MAFw project root.

Returns
Validated supported Python versions from `tool.mafw.supported-python`.

Return type
list[str]

Raises
DevtoolsError – If `pyproject.toml` is missing or does not identify MAFw.

mafw.devtools.dependencies.compile.**load_pylock_packages**(*pylock_path: Path*) → dict[str, dict[str, Any]]

Parse a pylock TOML file into a dictionary keyed by lowercase package name.

Each value in the returned dictionary contains at minimum the `name`, `version`, and optionally `marker` fields from the original TOML entry.

Parameters
pylock_path (*Path*) – Path to the pylock TOML file.

Returns
Dictionary mapping lowercase package names to their package metadata.

Return type
dict[str, dict[str, Any]]

Raises

- **FileNotFoundError** – If *pylock_path* does not exist.
- **tomllib.TOMLDecodeError** – If the file is not valid TOML.

mafw.devtools.dependencies.compile.**load_pyproject_doc**(*toml_text: str*) → TOMLDocument

Parse a `pyproject.toml` payload once and return the TOML document.

Parameters
toml_text (*str*) – Raw TOML payload.

Returns
Parsed TOML document.

Return type
tomlkit.TOMLDocument

Raises
DevtoolsError – If the TOML payload cannot be parsed.

mafw.devtools.dependencies.compile.**parse_python_version**(*version: str*) → tuple[int, int]

Parse a Python version string in `major.minor` form.

Parameters
version (*str*) – Python version string.

Returns
Major/minor version tuple.

Return type
tuple[int, int]

Raises
DevtoolsError – If the version is invalid.

mafw.devtools.dependencies.compile.**project_python_versions**() → list[str]

Read the supported CPython versions from `tool.mafw.supported-python` list in pyproject.toml.

The field is expected to be a list of strings representing CPython major.minor versions. The helper validates each entry and returns a sorted, de-duplicated list so downstream callers have deterministic ordering.

Returns
Sorted list of supported CPython versions.

Return type
list[str]

Raises
DevtoolsError – If `pyproject.toml` cannot be parsed or the field contains unsupported values.

mafw.devtools.dependencies.compile.**project_python_versions_from_doc**(*doc: TOMLDocument*) → list[str]

Extract supported CPython versions from a parsed `pyproject.toml` document.

Parameters
doc (*tomlkit.TOMLDocument*) – Parsed TOML document.

Returns
Sorted list of supported CPython versions.

Return type
list[str]

Raises
DevtoolsError – If the `tool.mafw.supported-python` field is invalid.

`mafw.devtools.dependencies.compile.`**`python_versions_between`**(*min_python_ver: str*, *max_python_ver: str*, *supported_versions: list[str]*) → list[str]

Build the inclusive list of Python versions between two bounds.

The returned versions must also be present in `supported_versions`.

Parameters

- **min_python_ver** (*str*) – Minimum Python version.
- **max_python_ver** (*str*) – Maximum Python version.
- **supported_versions** (*list[str]*) – List of supported Python versions from project metadata.

Returns
Ordered list of version strings.

Return type
list[str]

`mafw.devtools.dependencies.compile.`**`read_compiled_dependency_versions`**(*pylock_text: str*) → dict[str, Version]

Read resolved dependency versions from a compiled lockfile payload.

The returned mapping stores the highest resolved version seen for each dependency name, normalized to lowercase for stable lookups.

Parameters
pylock_text (*str*) – Raw `pylock.py3_14.toml` content.

Returns
Mapping of package name to highest resolved version.

Return type
dict[str, Version]

Raises
DevtoolsError – If the TOML payload cannot be parsed.

`mafw.devtools.dependencies.compile.`**`DEFAULT_FREEZE_EXTRAS: Final[tuple[str, ...]] = ('seaborn', 'all-db', 'steering-gui')`**

Extras used when compiling dependency lockfiles for release freezing and compatibility checks.

`mafw.devtools.dependencies.compile.`**`PYPROJECT_FILE: Final[Path] = PosixPath('pyproject.toml')`**

Path to the TOML file containing the project dependencies.

mafw.devtools.dependencies.freeze

Dependency freezing and unfreezing utilities for MAFw.

This module provides functions for adding and removing computed upper-bound constraints in `pyproject.toml` dependency declarations. It is used by the release workflow to pin dependencies during release and unpin them afterwards.

Functions

compute_upper_bound(lower_bound)	Compute an upper bound for a dependency based on PEP 440 compatible-release philosophy.
format_requirement(requirement)	Serialize a packaging requirement object back to a PEP 508 compatible string.
freeze_dependencies(*, dry_run)	Freeze dependencies by adding computed upper bounds in `pyproject.toml`.
freeze_pyproject_toml(toml_text, *[, ...])	Freeze dependencies in a `pyproject.toml` payload by adding upper bounds.
freeze_requirement(requirement_text, *[, ...])	Add a computed upper bound to a requirement string, if eligible.
has_frozen_upper_bound(requirement)	Determine whether a requirement already contains an upper bound constraint.
highest_lower_bound(requirement)	Extract the highest lower-bound version from >= and > specifiers.
iter_specifiers(requirement)	Return a concrete list of specifiers for the given requirement.
summarize_freeze_changes(before, after)	Build a short summary for dry-run output.
unfreeze_dependencies(...)	Unfreeze dependencies by removing computed upper bounds in `pyproject.toml`.
unfreeze_pyproject_toml(toml_text, *[, ...])	Unfreeze dependencies in a `pyproject.toml` payload by removing computed upper bounds.
unfreeze_requirement(requirement_text)	Remove a computed upper bound from a requirement string, if it matches the computed rule.
update_dependency_list(dependencies, *, ...)	Update a TOML list of dependency strings in-place.
update_requirements_and_readme(*, dry_run)	Update the requirements RST files and README.rst.

mafw.devtools.dependencies.freeze.**compute_upper_bound**(*lower_bound: str*) → str

Compute an upper bound for a dependency based on PEP 440 compatible-release philosophy.

The rule implemented here is purposely conservative and mirrors the intent of compatible release clauses while remaining explicit:

- For major-versioned releases (`X.*` with `X > 0`), freeze to `<(X + 1)`.
- For `0.*` releases, freeze to `<0.(minor + 1)` (rolling compatibility during pre-1.0).

Parameters
lower_bound (*str*) – Version string used as the starting point for the freeze rule.

Returns
Upper-bound version string without operator.

Return type
str

Raises
DevtoolsError – If the version cannot be parsed.

mafw.devtools.dependencies.freeze.**format_requirement**(*requirement: Requirement*) → str

Serialize a packaging requirement object back to a PEP 508 compatible string.

Parameters
requirement (*Requirement*) – Parsed requirement instance.

Returns
PEP 508 requirement string.

Return type
str

mafw.devtools.dependencies.freeze.**freeze_dependencies**(*, *dry_run: bool*) → str

Freeze dependencies by adding computed upper bounds in `pyproject.toml`.

Parameters
dry_run (*bool*) – Whether command execution is disabled.

Returns
Original `pyproject.toml` content captured before freeze.

Return type
str

mafw.devtools.dependencies.freeze.**freeze_pyproject_toml**(*toml_text: str*, *, *resolved_versions: dict[str, Version] | None = None*, *doc: TOMLDocument | None = None*) → tuple[str, list[str]]

Freeze dependencies in a `pyproject.toml` payload by adding upper bounds.

The TOML structure is preserved via tomlkit; only dependency strings may be normalized.

When `resolved_versions` is provided, the function uses those compiled versions as the basis for upper-bound computation. Otherwise the declared lower bounds are used as a fallback.

Parameters

- **toml_text** (*str*) – Raw TOML file content.
- **resolved_versions** (*dict[str, Version] | None*) – Optional mapping of dependency names to resolved versions.
- **doc** (*tomlkit.TOMLDocument | None*) – Optional pre-parsed TOML document to reuse when available.

Returns
Updated TOML plus warnings.

Return type
tuple[str, list[str]]

Raises
DevtoolsError – If TOML parsing fails.

mafw.devtools.dependencies.freeze.**freeze_requirement**(*requirement_text: str*, *, *resolved_version: str | None = None*) → tuple[str, list[str]]

Add a computed upper bound to a requirement string, if eligible.

The function is intentionally conservative:

- URL-based requirements are skipped (cannot be version constrained).
- Requirements already containing <, <=, ~=, == or === are skipped.
- Requirements without a lower bound emit a warning and are left unchanged.
- When a resolved version is provided, the upper bound is computed from that version instead of the declared lower bound.

Parameters
requirement_text (*str*) – Raw PEP 508 requirement string.

Returns
Updated requirement text plus warnings.

Return type
tuple[str, list[str]]

Raises
DevtoolsError – If parsing fails.

mafw.devtools.dependencies.freeze.**has_frozen_upper_bound**(*requirement: Requirement*) → bool

Determine whether a requirement already contains an upper bound constraint.

Parameters
requirement (*Requirement*) – Parsed requirement instance.

Returns
True if the requirement is already frozen.

Return type
bool

mafw.devtools.dependencies.freeze.**highest_lower_bound**(*requirement: Requirement*) → str | None

Extract the highest lower-bound version from >= and > specifiers.

Parameters
requirement (*Requirement*) – Parsed requirement instance.

Returns
Highest lower bound version string, or None if missing.

Return type
str | None

Raises
DevtoolsError – If version parsing fails.

mafw.devtools.dependencies.freeze.**iter_specifiers**(*requirement: Requirement*) → list[Specifier]

Return a concrete list of specifiers for the given requirement.

Parameters
requirement (*Requirement*) – Parsed requirement instance.

Returns
List of specifier objects.

Return type
list[Specifier]

mafw.devtools.dependencies.freeze.**summarize_freeze_changes**(*before: str*, *after: str*) → str

Build a short summary for dry-run output.

Parameters

- **before** (*str*) – Original TOML content.
- **after** (*str*) – Updated TOML content.

Returns
Human-readable summary line.

Return type
str

mafw.devtools.dependencies.freeze.**unfreeze_dependencies**(*original_pyproject_toml: str*, ***, *dry_run: bool*) → None

Unfreeze dependencies by removing computed upper bounds in pyproject.toml.

Parameters
dry_run (*bool*) – Whether command execution is disabled.

mafw.devtools.dependencies.freeze.**unfreeze_pyproject_toml**(*toml_text: str*, ***, *baseline_toml_text: str | None = None*, *doc: TOMLDocument | None = None*, *baseline_doc: TOMLDocument | None = None*) → tuple[str, list[str]]

Unfreeze dependencies in a `pyproject.toml` payload by removing computed upper bounds.

Only upper bounds matching the computed rule are removed; existing manual constraints remain.

Parameters

- **toml_text** (*str*) – Raw TOML file content.
- **baseline_toml_text** (*str | None*) – Optional original TOML text captured before freezing. When provided, unfreezing is computed against the baseline to avoid altering dependencies that were already frozen before release.
- **doc** (*tomlkit.TOMLDocument | None*) – Optional pre-parsed TOML document to reuse when available.
- **baseline_doc** (*tomlkit.TOMLDocument | None*) – Optional pre-parsed baseline TOML document to reuse when available.

Returns
Updated TOML plus warnings.

Return type
tuple[str, list[str]]

Raises
DevtoolsError – If TOML parsing fails.

mafw.devtools.dependencies.freeze.**unfreeze_requirement**(*requirement_text: str*) → tuple[str, list[str]]

Remove a computed upper bound from a requirement string, if it matches the computed rule.

Only the auto-generated upper bound <upper is removed; existing manual upper bounds are preserved.

Parameters
requirement_text (*str*) – Raw PEP 508 requirement string.

Returns
Updated requirement text plus warnings.

Return type
tuple[str, list[str]]

Raises
DevtoolsError – If parsing fails.

mafw.devtools.dependencies.freeze.**update_dependency_list**(*dependencies: Any*, *, *transformer: Callable[[str], tuple[str, list[str]]]*, *warnings: list[str]*, *context: str*) → None

Update a TOML list of dependency strings in-place.

Parameters

- **dependencies** (*Any*) – TOML array that contains dependency strings.
- **transformer** (*Any*) – Callable applied to each dependency string.
- **warnings** (*list[str]*) – List of warnings to append to.
- **context** (*str*) – Human-readable location for warnings.

mafw.devtools.dependencies.freeze.**update_requirements_and_readme**(*, *dry_run: bool*) → None

Update the requirements RST files and README.rst.

Parameters
dry_run (*bool*) – Whether command execution is disabled.

mafw.devtools.dependencies.freeze.**_FROZEN_OPERATORS: Final[set[str]] = {'<', '<=', '==', '===', '~='}**

Operators that already constrain the maximum compatible version and should not be auto-frozen.

mafw.devtools.dependencies.verify

Dependency resolution verification utilities for MAFw.

This module provides functions to verify that resolved dependency environments match the expected lower bounds declared in `pyproject.toml`.

Functions

get_expected_lower_bounds()	Read pyproject.toml and extract dependencies with their expected lower bounds.
verify_lowest_resolution(env_name, ...)	Verify that the specified environment has the expected lowest dependency versions.

mafw.devtools.dependencies.verify.**get_expected_lower_bounds**() → dict[str, Requirement]

Read pyproject.toml and extract dependencies with their expected lower bounds.

Returns
A dictionary mapping lowercase package names to their parsed Requirement objects.

Return type
dict[str, Requirement]

Raises
DevtoolsError – If the pyproject.toml cannot be parsed.

mafw.devtools.dependencies.verify.**verify_lowest_resolution**(*env_name: str*, *python_version: str*, *expected_bounds: dict[str, Requirement]*) → None

Verify that the specified environment has the expected lowest dependency versions.

This function executes *uv pip list* in the given environment, parses the output, and compares installed versions against the expected lower bounds extracted from *pyproject.toml*.

Parameters

- **env_name** (*str*) – The name of the environment (e.g., 'hatch-test', 'types').
- **python_version** (*str*) – The Python version string (e.g., '3.11').
- **expected_bounds** (*dict[str, Requirement]*) – Mapping of package names to their Requirement objects.

Raises
DevtoolsError – If a required dependency is missing.

mafw.devtools.documentation

Documentation management utilities for MAFw development tools.

This package provides business logic for building versioned documentation, managing versions.json, generating redirect pages, creating requirements documentation, and running local documentation servers.

class mafw.devtools.documentation.**ServerStatusEnum**(**values*)

Bases: `StrEnum`

Possible status values for the multiversion-doc local documentation server.

Variables

- **running** – The server process exists, matches the expected signature and responds over HTTP.
- **stale** – The server process exists and matches the expected signature, but does not respond over HTTP.
- **stopped** – The recorded PID does not exist or does not match the expected signature.
- **unknown** – The metadata file is missing, so no status can be determined.

static _generate_next_value_(*name*, *start*, *count*, *last_values*)

Return the lower-cased version of the member name.

mafw.devtools.documentation.**_load_supported_python_versions**(*, *repo_root: Path | None = None*) → list[str]

Load the supported Python versions declared in `pyproject.toml`.

Parameters
repo_root (*Path | None*) – Repository root directory, defaults to auto-detection

Returns
Sorted list of supported `major.minor` versions.

Return type
list[str]

mafw.devtools.documentation.**build_for_tag**(*tag: str*, *outdir: Path*, *tmproot: Path*, *use_latest_conf: bool = False*, *keep_tmp: bool = False*) → tuple[bool, str]

Create worktree for tag, run sphinx-build, save log.

Parameters

- **tag** (*str*) – Git tag to build documentation for
- **outdir** (*Path*) – Output directory for built documentation
- **tmproot** (*Path*) – Root temporary directory
- **use_latest_conf** (*bool*) – Whether to use latest conf.py, defaults to False
- **keep_tmp** (*bool*) – Whether to keep temporary files, defaults to False

Returns
Tuple of (success, log_contents)

Return type
tuple[bool, str]

mafw.devtools.documentation.**build_gitlab_api_configuration**(*api_url: str | None*, *project_id: int | None*, *token: str | None*) → *GitlabAPIConfiguration*

Build a GitLab API configuration from provided values and environment context.

Parameters

- **api_url** (*str | None*) – Optional override for the GitLab API URL.
- **project_id** (*int | None*) – Optional override for the GitLab project ID.
- **token** (*str | None*) – Optional override for the authentication token.

Returns
The validated GitLab API configuration.

Return type
gitlab.GitlabAPIConfiguration

Raises
ValueError – If any required configuration value is missing.

mafw.devtools.documentation.**build_gitlab_auth_headers**(*api_config: GitlabAPIConfiguration*) → MutableMapping[str, str | bytes]

Build authentication headers for GitLab API requests.

Parameters
api_config (gitlab.GitlabAPIConfiguration) – The GitLab API configuration.

Returns
A mutable mapping of HTTP headers.

Return type
MutableMapping[str, str | bytes]

mafw.devtools.documentation.**build_pdf_for_tag**(*tag: str*, *html_tag_dir: Path*, *tmproot: Path*, *use_latest_conf: bool = False*, *keep_tmp: bool = False*) → tuple[bool, str, Path | None]

Create worktree for tag, run sphinx-build with latex builder, then make PDF.

Parameters

- **tag** (*str*) – Git tag to build PDF for
- **html_tag_dir** (*Path*) – Directory containing HTML output for the tag
- **tmproot** (*Path*) – Root temporary directory
- **use_latest_conf** (*bool*) – Whether to use latest conf.py, defaults to False
- **keep_tmp** (*bool*) – Whether to keep temporary files, defaults to False

Returns
Tuple of (success, log_contents, pdf_path)

Return type
tuple[bool, str, Path | None]

mafw.devtools.documentation.**check_multiversion_structure**(*outdir: Path*) → bool

Check if multiversion structure exists (other version directories).

Parameters
outdir (*Path*) – Output directory to check

Returns
True if other versions exist

Return type
bool

mafw.devtools.documentation.**copy_patch_files**(*docs_src: Path*) → None

Copy patch files needed for older versions.

Parameters
docs_src (*Path*) – Path to documentation source directory

mafw.devtools.documentation.**create_docs_zip_for_tag**(*outdir: Path*, *tag: str*, *zip_filepath: Path*) → Path

Create a zip archive for a built documentation version directory.

The built docs are expected under `outdir / tag` (e.g. `docs/build/doc/vX.Y.Z`). The produced zip is laid out so that extracting it from the repository root recreates the original directory structure (e.g. `docs/build/doc/vX.Y.Z/...`).

The zip file is created at `zip_filepath / f"mafw-docs-{tag}.zip"`.

Notes

- Symlinks are skipped to avoid ambiguous extraction behavior across platforms.

param outdir
Output directory that contains the built docs

type outdir
Path

param tag
Version tag (e.g. `v2.1.0`)

type tag
str

param zip_filepath
Directory where the zip file is written

type zip_filepath
Path

return
Path to the created zip archive

rtype
Path

raises FileNotFoundError
If the built docs directory does not exist

raises NotADirectoryError
If the built docs path is not a directory

mafw.devtools.documentation.**download_docs_zip_from_gitlab_generic_registry**(*api_config: GitlabAPIConfiguration*, *package_version: str*, *download_dir: Path*, *package_name: str = 'mafw-docs'*, *file_name: str | None = None*, *timeout_s: float = 60.0*) → Path | None

Download a documentation zip archive from the GitLab Generic Package Registry.

The equivalent GitLab API endpoint is:

`GET /projects/:id/packages/generic/:package_name/:package_version/:file_name`

The function first issues a HEAD request to determine if the file exists: - 404: the package file does not exist (cache miss) and `None` is returned. - 200: the file exists and is downloaded. - otherwise: a warning is printed and `None` is returned.

Authentication headers: - `JOB-TOKEN` when running on CI (`api_config.on_ci` is True) - `PRIVATE-TOKEN` for local execution

Parameters

- **api_config** (gitlab.GitlabAPIConfiguration) – GitLab API configuration
- **package_version** (*str*) – Package version (typically the git tag, e.g. `v2.1.0`)
- **download_dir** (*Path*) – Directory where the downloaded zip is stored
- **package_name** (*str*) – Generic package name, defaults to `mafw-docs`
- **file_name** (*str | None*) – File name to retrieve, defaults to `mafw-docs-<version>.zip`
- **timeout_s** (*float*) – Request timeout in seconds, defaults to 60.0

Returns
Path to the downloaded zip archive, or None if it does not exist or cannot be retrieved

Return type
Path | None

mafw.devtools.documentation.**ensure_sphinx_build_available**() → None

Ensure that the Sphinx Python package is available.

`devtools documentation` is a development helper shipped with MAFw. The script is typically executed from the optional `[dev]` environment (either by activating the development environment and using the console entry point, or via `hatch run dev.py<version>:devtools documentation` on CI/CD).

Checking for the `sphinx-build` executable alone is not sufficient because the effective availability depends on which Python environment is executing the command. Checking the import spec for `sphinx` validates that the correct optional dependencies are installed for the running interpreter.

Raises
DevtoolsError – If Sphinx is not available.

mafw.devtools.documentation.**ensure_versions_json_exists**(*outdir: Path*) → bool

Ensure versions.json exists in outdir. If not, try to copy from another version.

Parameters
outdir (*Path*) – Output directory that should contain versions.json

Returns
True if versions.json exists or was successfully copied

Return type
bool

mafw.devtools.documentation.**extract_docs_zip_to_repo_root**(*zip_path: Path*, *repo_root: Path | None = None*) → None

Extract a documentation zip archive into the repository root and remove the zip.

The zip archive is expected to contain paths rooted at the repository (e.g. `docs/build/doc/vX.Y.Z/...`) so that extraction recreates the same structure as a normal documentation build.

Parameters

- **zip_path** (*Path*) – Zip archive to extract
- **repo_root** (*Path | None*) – Repository root directory, defaults to auto-detection

Raises

- **FileNotFoundError** – If `zip_path` does not exist
- **zipfile.BadZipFile** – If the archive is invalid

mafw.devtools.documentation.**filter_latest_micro**(*versions: list[tuple[Version, Any]]*) → list[tuple[Version, Any]]

Keep only the latest micro version per minor (major.minor).

Parameters
versions (*list[tuple[Version, Any]]*) – List of (Version, tag) tuples

Returns
Filtered list of (Version, tag) tuples

Return type
list[tuple[Version, Any]]

mafw.devtools.documentation.**filter_stable_tags**(*tags: list[str]*, *regex: str*) → list[str]

Filter tags based on a regular expression pattern.

Parameters

- **tags** (*list[str]*) – List of tag strings to filter
- **regex** (*str*) – Regular expression pattern to match against

Returns
Filtered list of matching tags

Return type
list[str]

mafw.devtools.documentation.**filter_versions_in_range**(*versions: list[str]*, *from_v: str | None*, *to_v: str | None*) → list[str]

Filter versions within an inclusive semantic-version range.

Parameters

- **versions** (*list[str]*) – Input versions list
- **from_v** (*str | None*) – Range start (inclusive)
- **to_v** (*str | None*) – Range end (inclusive)

Returns
Filtered versions list

Return type
list[str]

Raises
ValueError – If range bounds are invalid

mafw.devtools.documentation.**find_repo_root**(*start: Path | None = None*) → Path

Find the repository root directory.

The root is detected by walking upwards until a `pyproject.toml` file is found. If no such file is found, the starting directory is returned.

Parameters
start (*Path | None*) – Directory from which to start searching, defaults to current working directory

Returns
Resolved repository root directory

Return type
Path

mafw.devtools.documentation.**format_size**(*bytes_size: float*) → str

Format bytes to human-readable size.

Parameters
bytes_size (*int*) – Size in bytes

Returns
Formatted size string

Return type
str

mafw.devtools.documentation.**format_started_at_utc**(*now: datetime*) → str

Format a timestamp for server metadata.

Parameters
now (*datetime*) – A datetime instance (timezone-aware is preferred)

Returns
UTC ISO-8601 timestamp with Z suffix

Return type
str

mafw.devtools.documentation.**generate_pdf_index_page**(*html_outdir: Path*, *pdf_info: list[dict[str, str]]*, *project_name: str = 'Documentation'*) → None

Generate an HTML page listing all available PDFs. This page will be placed in the root html_versions directory. Order: stable first, then latest, then other releases sorted by version (newest first).

Parameters

- **html_outdir** (*Path*) – Output directory for HTML files
- **pdf_info** (*list[dict[str, str]]*) – List of dictionaries containing PDF information
- **project_name** (*str*) – Name of the project for the page title, defaults to 'Documentation'

mafw.devtools.documentation.**generate_python_versions_rst**(*, *repo_root: Path | None = None*) → None

Generate the RST substitution file for the supported Python range.

The file is emitted under `docs/source/requirements` so it can be included by the general documentation and copied into the README update block.

Parameters
repo_root (*Path | None*) – Repository root directory, defaults to auto-detection

mafw.devtools.documentation.**generate_requirements_rst**(*group_name: str = 'base'*, *, *repo_root: Path | None = None*) → None

Generate an RST file with the dependencies of a given group.

The function parses pyproject.toml to retrieve the dependencies and their descriptions from the tool.mafw.dependency-description section. The generated file is saved as <group_name>_requirements.rst in the docs/source directory.

Parameters

- **group_name** (*str*) – The name of the dependency group (e.g., 'base', 'seaborn'), defaults to 'base'
- **repo_root** (*Path | None*) – Repository root directory, defaults to auto-detection

mafw.devtools.documentation.**get_current_branch**() → str

Get the name of the currently checked out branch.

Returns
Name of the current branch

Return type
str

mafw.devtools.documentation.**get_directory_size**(*path: Path*) → int

Calculate total size of a directory in bytes.

Parameters
path (*Path*) – Directory path

Returns
Total size in bytes

Return type
int

mafw.devtools.documentation.**get_git_tags**(*min_version: str | None = None*) → list[tuple[Version, Any]]

Return list of (Version, tag) tuples sorted ascending.

Parameters
min_version (*str | None*) – Minimum version to consider, defaults to None

Returns
List of (Version, tag) tuples

Return type
list[tuple[Version, Any]]

mafw.devtools.documentation.**git_rev_of**(*ref: str*) → str

Get the git revision hash for a given reference.

Parameters
ref (*str*) – Git reference (tag, branch, commit hash)

Returns
Git revision hash

Return type
str

Raises
RuntimeError – If git rev-list fails

mafw.devtools.documentation.**http_probe**(*address: str*, *port: int*, *timeout_s: float = 1.0*) → bool

Probe whether the server responds at the given address:port.

Parameters

- **address** (*str*) – Server address
- **port** (*int*) – Server port
- **timeout_s** (*float*) – Timeout in seconds

Returns
True if a GET to the server root returns HTTP 200

Return type
bool

mafw.devtools.documentation.**is_ancestor**(*a: str*, *b: str*) → bool

Return True if commit a is ancestor of commit b (git merge-base –is-ancestor).

Parameters

- **a** (*str*) – First commit reference
- **b** (*str*) – Second commit reference

Returns
True if a is ancestor of b

Return type
bool

mafw.devtools.documentation.**iter_local_mafw_docs_zips**(*zip_dir: Path*) → list[tuple[str, Path]]

List local mafw-docs zip files in a directory.

Only files matching `mafw-docs-vX.Y.Z.zip` are returned.

Parameters
zip_dir (*Path*) – Directory to scan

Returns
List of (version, file_path) tuples

Return type
list[tuple[str, Path]]

mafw.devtools.documentation.**list_mafw_docs_generic_packages**(*api_config: GitlabAPIConfiguration*, *package_name: str = 'mafw-docs'*) → list[dict[str, Any]]

List generic packages for mafw-docs in the GitLab Package Registry.

Uses the Packages API: `GET /projects/:id/packages` and filters for `package_type=generic` and exact `name == package_name`.

Parameters

- **api_config** (gitlab.GitlabAPIConfiguration) – GitLab API configuration
- **package_name** (*str*) – Package name, defaults to `mafw-docs`

Returns
List of package dictionaries from the API

Return type
list[dict[str, Any]]

mafw.devtools.documentation.**mirror_version**(*outdir: Path*, *src_tag: str*, *target_tag: str*, *use_symlink: bool = True*) → None

Mirror a version directory from one tag to another. Can use symlinks for efficiency or copy for compatibility.

Parameters

- **outdir** (*Path*) – Output directory containing version directories
- **src_tag** (*str*) – Source tag directory name
- **target_tag** (*str*) – Target tag directory name
- **use_symlink** (*bool*) – Whether to use symlink instead of copying, defaults to True

mafw.devtools.documentation.**normalize_registry_item**(*item: str*) → tuple[str, str]

Normalize a registry item into (version, file_name).

The item can be either: - a version string: `vX.Y.Z` - a file name: `mafw-docs-vX.Y.Z.zip`

Parameters
item (*str*) – Input item

Returns
Tuple of (version, file_name)

Return type
tuple[str, str]

Raises
ValueError – If the item cannot be normalized

mafw.devtools.documentation.**parse_mafw_docs_zip_filename**(*file_name: str*) → tuple[str, str] | None

Parse and validate a mafw-docs zip filename.

The accepted filename pattern is: `mafw-docs-vX.Y.Z.zip`.

Parameters
file_name (*str*) – File name to parse

Returns
Tuple of (version, normalized_file_name) if valid, otherwise None

Return type
tuple[str, str] | None

mafw.devtools.documentation.**parse_sphinx_log**(*log_content: str*) → tuple[int, int, list[str]]

Parse Sphinx build log to extract warning and error counts, and warning messages.

Only three warnings are reported

Parameters
log_content (*str*) – Sphinx build log

Returns
Tuple of warning, error count, warning messages

Return type
tuple[int, int, list[str]]

mafw.devtools.documentation.**parse_version_tuple**(*tag: str*) → tuple[int, ...]

Parse vX.Y.Z(.W) into tuple of ints for sorting.

Parameters
tag (*str*) – Version tag string

Returns
Tuple of integers representing the version

Return type
tuple[int, ...]

mafw.devtools.documentation.**process_matches_http_server**(*proc: Any*, *address: str*, *port: int*, *directory: Path*) → bool

Check whether a process resembles the expected http.server invocation.

Parameters

- **proc** (*Any*) – psutil Process instance
- **address** (*str*) – Expected bind address
- **port** (*int*) – Expected port
- **directory** (*Path*) – Expected served directory

Returns
True when the process signature matches

Return type
bool

mafw.devtools.documentation.**prune_old_versions**(*outdir: Path*, *max_size_mb: int = 100*, *dry_run: bool = False*) → tuple[list[str], int]

Remove oldest version directories until total size is below threshold. Always keeps 'stable', 'latest', and 'dev' (if present).

Parameters

- **outdir** (*Path*) – Output directory containing version directories

- **max_size_mb** (*int*) – Maximum size in megabytes
- **dry_run** (*bool*) – If True, only report what would be deleted

Returns
Tuple of (list of removed versions, final size in bytes)

Return type
tuple[list[str], int]

mafw.devtools.documentation.**read_server_metadata**(*directory: Path*) → dict[str, Any] | None

Read local documentation server metadata if available.

Parameters
directory (*Path*) – Served directory to locate metadata under

Returns
Parsed metadata, or None when the metadata file is missing

Return type
dict[str, Any] | None

Raises
DevtoolsError – If the metadata file exists but cannot be parsed

mafw.devtools.documentation.**regenerate_versions_json_after_pruning**(*outdir: Path*, *removed_versions: list[str]*) → None

Regenerate versions.json after pruning, excluding removed versions.

Parameters

- **outdir** (*Path*) – Output directory containing version directories
- **removed_versions** (*list[str]*) – List of version names that were removed

mafw.devtools.documentation.**report_build_status**(*tag: str*, *success: bool*, *log: str*, *build_type: str = 'HTML'*) → None

Report build status with warning/error summary.

Parameters

- **tag** (*str*) – Version tag being built
- **success** (*bool*) – Whether build succeeded
- **log** (*str*) – Build log content
- **build_type** (*str*) – Type of build (HTML or PDF)

mafw.devtools.documentation.**resolve_mafw_docs_package_ids_by_version**(*api_config: GitlabAPIConfiguration*, *package_name: str = 'mafw-docs'*) → dict[str, int]

Resolve package IDs for mafw-docs generic packages by version.

Parameters

- **api_config** (gitlab.GitlabAPIConfiguration) – GitLab API configuration
- **package_name** (*str*) – Package name, defaults to `mafw-docs`

Returns
Mapping from version string to package id

Return type
dict[str, int]

mafw.devtools.documentation.**run**(*cmd: list[str]*, *cwd: Path | None = None*) → CompletedProcess[str]

Helper to run commands with consistent behavior.

Parameters

- **cmd** (*list[str]*) – Command to execute as a list of strings
- **cwd** (*Path | None*) – Working directory for command execution, defaults to None

Returns
Completed process result

Return type
subprocess.CompletedProcess[str]

mafw.devtools.documentation.**server_default_directory**() → Path

Return the default directory served by the local documentation server.

Returns
Default documentation build directory

Return type
Path

mafw.devtools.documentation.**server_metadata_path**(*directory: Path*) → Path

Return the metadata path associated to a served directory.

Parameters
directory (*Path*) – Served directory

Returns
Metadata JSON file path

Return type
Path

mafw.devtools.documentation.**sort_tags_semver**(*tags: list[str]*) → list[str]

Sort tags using semantic versioning comparison.

Parameters
tags (*list[str]*) – List of tag strings to sort

Returns
Sorted list of tag strings

Return type
list[str]

mafw.devtools.documentation.**upload_docs_zip_to_gitlab_generic_registry**(*api_config: GitlabAPIConfiguration*, *package_version: str*, *zip_path: Path*, *package_name: str = 'mafw-docs'*, *timeout_s: float = 60.0*) → bool

Upload a documentation zip archive to the GitLab Generic Package Registry.

The equivalent GitLab API endpoint is:

`PUT /projects/:id/packages/generic/:package_name/:package_version/:file_name`

Authentication headers: - `JOB-TOKEN` when running on CI (`api_config.on_ci` is True) - `PRIVATE-TOKEN` for local execution

Parameters

- **api_config** (gitlab.GitlabAPIConfiguration) – GitLab API configuration
- **package_version** (*str*) – Package version (typically the git tag, e.g. `v2.1.0`)
- **zip_path** (*Path*) – Path to the zip file to upload
- **package_name** (*str*) – Generic package name, defaults to `mafw-docs`
- **timeout_s** (*float*) – Request timeout in seconds, defaults to 60.0

Returns
True if the file was uploaded, False if the upload was skipped because the target already exists

Return type
bool

Raises

- **FileNotFoundError** – If `zip_path` does not exist
- **RuntimeError** – If the upload fails (non-2xx response)

mafw.devtools.documentation.**write_legacy_redirect_page**(*outdir: Path*) → None

Create a legacy redirect page at the root of the output directory.

Parameters
outdir (*Path*) – Output directory for the redirect page

mafw.devtools.documentation.**write_redirect_page**(*outdir: Path*, *name: str*, *target_tag: str*) → None

Create a redirect page for a version alias.

Parameters

- **outdir** (*Path*) – Output directory for the redirect page
- **name** (*str*) – Name of the redirect alias (e.g., 'stable', 'dev')
- **target_tag** (*str*) – Tag that the redirect should point to

mafw.devtools.documentation.**write_redirects_file**(*outdir: Path*) → None

Create a _redirects file for GitLab Pages.

Parameters
outdir (*Path*) – Output directory for the redirects file

mafw.devtools.documentation.**write_root_landing_page**(*build_root: Path*, *project_name: str = 'MAFw'*) → None

Create a landing page for the project root with links to documentation and coverage.

Parameters

- **build_root** (*Path*) – Root build directory (should contain 'doc' subdirectory)
- **project_name** (*str*) – Project name for the page title

mafw.devtools.documentation.**write_server_metadata**(*directory: Path*, *metadata: dict[str, Any]*) → None

Write local documentation server metadata.

Parameters

- **directory** (*Path*) – Served directory where metadata is stored
- **metadata** (*dict[str, Any]*) – Metadata dictionary to serialize as JSON

Raises
DevtoolsError – If the metadata file cannot be written

mafw.devtools.documentation.**write_versions_json**(*outdir: Path*, *versions: list[dict[str, str]]*) → None

Write versions information to a JSON file.

Parameters

- **outdir** (*Path*) – Output directory for the JSON file
- **versions** (*list[dict[str, str]]*) – List of version information dictionaries

Modules

builder	Sphinx documentation building helpers for MAFw versioned documentation.
requirements	Requirements documentation constants and generators for MAFw.
server	Local documentation server management for MAFw.
versions	Version management helpers for MAFw versioned documentation.

mafw.devtools.documentation.builder

Sphinx documentation building helpers for MAFw versioned documentation.

This module provides functions for building Sphinx documentation across multiple git tags, managing git worktrees, and handling documentation zip archives.

Module Attributes

DEFAULT_MIN_TAG_REGEX	Regular expression to match stable version tags.
DOCS_SUBPATH	The files/directories under each worktree where docs live.
SPHINX_BUILD_CMD	Sphinx build command name.
OLD_VERSION_TO_BE_PATCHED	Tags that require patching with the latest conf.py.

Functions

build_for_tag(tag, outdir, tmproot[, ...])	Create worktree for tag, run sphinx-build, save log.
build_pdf_for_tag(tag, html_tag_dir, tmproot)	Create worktree for tag, run sphinx-build with latex builder, then make PDF.
check_multiversion_structure(outdir)	Check if multiversion structure exists (other version directories).
copy_patch_files(docs_src)	Copy patch files needed for older versions.
create_docs_zip_for_tag(outdir, tag, ...)	Create a zip archive for a built documentation version directory.
ensure_sphinx_build_available()	Ensure that the Sphinx Python package is available.
extract_docs_zip_to_repo_root(zip_path[, ...])	Extract a documentation zip archive into the repository root and remove the zip.
filter_latest_micro(versions)	Keep only the latest micro version per minor (major.minor).
filter_stable_tags(tags, regex)	Filter tags based on a regular expression pattern.
filter_versions_in_range(versions, from_v, to_v)	Filter versions within an inclusive semantic-version range.
find_repo_root([start])	Find the repository root directory.
generate_pdf_index_page(html_outdir, pdf_info)	Generate an HTML page listing all available PDFs.

continues on next page

Table 14.51 – continued from previous page

iter_local_mafw_docs_zips(zip_dir)	List local mafw-docs zip files in a directory.
normalize_registry_item(item)	Normalize a registry item into (version, file_name).
parse_mafw_docs_zip_filename(file_name)	Parse and validate a mafw-docs zip filename.
parse_sphinx_log(log_content)	Parse Sphinx build log to extract warning and error counts, and warning messages.
parse_version_tuple(tag)	Parse vX.Y.Z(.W) into tuple of ints for sorting.
report_build_status(tag, success, log[, ...])	Report build status with warning/error summary.
run(cmd[, cwd])	Helper to run commands with consistent behavior.

mafw.devtools.documentation.builder.**build_for_tag**(*tag: str*, *outdir: Path*, *tmproot: Path*, *use_latest_conf: bool = False*, *keep_tmp: bool = False*) → tuple[bool, str]

Create worktree for tag, run sphinx-build, save log.

Parameters

- **tag** (*str*) – Git tag to build documentation for
- **outdir** (*Path*) – Output directory for built documentation
- **tmproot** (*Path*) – Root temporary directory
- **use_latest_conf** (*bool*) – Whether to use latest conf.py, defaults to False
- **keep_tmp** (*bool*) – Whether to keep temporary files, defaults to False

Returns
Tuple of (success, log_contents)

Return type
tuple[bool, str]

mafw.devtools.documentation.builder.**build_pdf_for_tag**(*tag: str*, *html_tag_dir: Path*, *tmproot: Path*, *use_latest_conf: bool = False*, *keep_tmp: bool = False*) → tuple[bool, str, Path | None]

Create worktree for tag, run sphinx-build with latex builder, then make PDF.

Parameters

- **tag** (*str*) – Git tag to build PDF for
- **html_tag_dir** (*Path*) – Directory containing HTML output for the tag
- **tmproot** (*Path*) – Root temporary directory
- **use_latest_conf** (*bool*) – Whether to use latest conf.py, defaults to False
- **keep_tmp** (*bool*) – Whether to keep temporary files, defaults to False

Returns
Tuple of (success, log_contents, pdf_path)

Return type
tuple[bool, str, Path | None]

mafw.devtools.documentation.builder.**check_multiversion_structure**(*outdir: Path*) → bool

Check if multiversion structure exists (other version directories).

Parameters
outdir (*Path*) – Output directory to check

Returns
True if other versions exist

Return type
bool

mafw.devtools.documentation.builder.**copy_patch_files**(*docs_src: Path*) → None

Copy patch files needed for older versions.

Parameters
docs_src (*Path*) – Path to documentation source directory

mafw.devtools.documentation.builder.**create_docs_zip_for_tag**(*outdir: Path*, *tag: str*, *zip_filepath: Path*) → Path

Create a zip archive for a built documentation version directory.

The built docs are expected under `outdir / tag` (e.g. `docs/build/doc/vX.Y.Z`). The produced zip is laid out so that extracting it from the repository root recreates the original directory structure (e.g. `docs/build/doc/vX.Y.Z/...`).

The zip file is created at `zip_filepath / f"mafw-docs-{tag}.zip"`.

Notes

- Symlinks are skipped to avoid ambiguous extraction behavior across platforms.

param outdir
Output directory that contains the built docs

type outdir
Path

param tag
Version tag (e.g. `v2.1.0`)

type tag
str

param zip_filepath
Directory where the zip file is written

type zip_filepath
Path

return
Path to the created zip archive

rtype
Path

raises FileNotFoundError
If the built docs directory does not exist

raises NotADirectoryError
If the built docs path is not a directory

mafw.devtools.documentation.builder.**ensure_sphinx_build_available**() → None

Ensure that the Sphinx Python package is available.

`devtools documentation` is a development helper shipped with MAFw. The script is typically executed from the optional `[dev]` environment (either by activating the development environment and using the console entry point, or via `hatch run dev.py<version>:devtools documentation` on CI/CD).

Checking for the `sphinx-build` executable alone is not sufficient because the effective availability depends on which Python environment is executing the command. Checking the import spec for `sphinx` validates that the correct optional dependencies are installed for the running interpreter.

Raises
DevtoolsError – If Sphinx is not available.

mafw.devtools.documentation.builder.**extract_docs_zip_to_repo_root**(*zip_path: Path*, *repo_root: Path | None = None*) → None

Extract a documentation zip archive into the repository root and remove the zip.

The zip archive is expected to contain paths rooted at the repository (e.g. `docs/build/doc/vX.Y.Z/...`) so that extraction recreates the same structure as a normal documentation build.

Parameters

- **zip_path** (*Path*) – Zip archive to extract
- **repo_root** (*Path | None*) – Repository root directory, defaults to auto-detection

Raises

- **FileNotFoundError** – If `zip_path` does not exist
- **zipfile.BadZipFile** – If the archive is invalid

mafw.devtools.documentation.builder.**filter_latest_micro**(*versions: list[tuple[Version, Any]]*) → list[tuple[Version, Any]]

Keep only the latest micro version per minor (major.minor).

Parameters
versions (*list[tuple[Version, Any]]*) – List of (Version, tag) tuples

Returns
Filtered list of (Version, tag) tuples

Return type
list[tuple[Version, Any]]

mafw.devtools.documentation.builder.**filter_stable_tags**(*tags: list[str]*, *regex: str*) → list[str]

Filter tags based on a regular expression pattern.

Parameters

- **tags** (*list[str]*) – List of tag strings to filter
- **regex** (*str*) – Regular expression pattern to match against

Returns
Filtered list of matching tags

Return type
list[str]

mafw.devtools.documentation.builder.**filter_versions_in_range**(*versions: list[str]*, *from_v: str | None*, *to_v: str | None*) → list[str]

Filter versions within an inclusive semantic-version range.

Parameters

- **versions** (*list[str]*) – Input versions list
- **from_v** (*str | None*) – Range start (inclusive)
- **to_v** (*str | None*) – Range end (inclusive)

Returns
Filtered versions list

Return type
list[str]

Raises
ValueError – If range bounds are invalid

mafw.devtools.documentation.builder.**find_repo_root**(*start: Path | None = None*) → Path

Find the repository root directory.

The root is detected by walking upwards until a `pyproject.toml` file is found. If no such file is found, the starting directory is returned.

Parameters
start (*Path | None*) – Directory from which to start searching, defaults to current working directory

Returns
Resolved repository root directory

Return type
Path

mafw.devtools.documentation.builder.**generate_pdf_index_page**(*html_outdir: Path*, *pdf_info: list[dict[str, str]]*, *project_name: str = 'Documentation'*) → None

Generate an HTML page listing all available PDFs. This page will be placed in the root html_versions directory. Order: stable first, then latest, then other releases sorted by version (newest first).

Parameters

- **html_outdir** (*Path*) – Output directory for HTML files
- **pdf_info** (*list[dict[str, str]]*) – List of dictionaries containing PDF information
- **project_name** (*str*) – Name of the project for the page title, defaults to 'Documentation'

mafw.devtools.documentation.builder.**iter_local_mafw_docs_zips**(*zip_dir: Path*) → list[tuple[str, Path]]

List local mafw-docs zip files in a directory.

Only files matching `mafw-docs-vX.Y.Z.zip` are returned.

Parameters
zip_dir (*Path*) – Directory to scan

Returns
List of (version, file_path) tuples

Return type
list[tuple[str, Path]]

mafw.devtools.documentation.builder.**normalize_registry_item**(*item: str*) → tuple[str, str]

Normalize a registry item into (version, file_name).

The item can be either: - a version string: `vX.Y.Z` - a file name: `mafw-docs-vX.Y.Z.zip`

Parameters
item (*str*) – Input item

Returns
Tuple of (version, file_name)

Return type
tuple[str, str]

Raises
ValueError – If the item cannot be normalized

mafw.devtools.documentation.builder.**parse_mafw_docs_zip_filename**(*file_name: str*) → tuple[str, str] | None

Parse and validate a mafw-docs zip filename.

The accepted filename pattern is: `mafw-docs-vX.Y.Z.zip`.

Parameters
file_name (*str*) – File name to parse

Returns
Tuple of (version, normalized_file_name) if valid, otherwise None

Return type
tuple[str, str] | None

mafw.devtools.documentation.builder.**parse_sphinx_log**(*log_content: str*) → tuple[int, int, list[str]]

Parse Sphinx build log to extract warning and error counts, and warning messages.

Only three warnings are reported

Parameters
log_content (*str*) – Sphinx build log

Returns
Tuple of warning, error count, warning messages

Return type
tuple[int, int, list[str]]

mafw.devtools.documentation.builder.**parse_version_tuple**(*tag: str*) → tuple[int, ...]

Parse vX.Y.Z(.W) into tuple of ints for sorting.

Parameters
tag (*str*) – Version tag string

Returns
Tuple of integers representing the version

Return type
tuple[int, ...]

mafw.devtools.documentation.builder.**report_build_status**(*tag: str*, *success: bool*, *log: str*, *build_type: str = 'HTML'*) → None

Report build status with warning/error summary.

Parameters

- **tag** (*str*) – Version tag being built
- **success** (*bool*) – Whether build succeeded
- **log** (*str*) – Build log content
- **build_type** (*str*) – Type of build (HTML or PDF)

mafw.devtools.documentation.builder.**run**(*cmd: list[str]*, *cwd: Path | None = None*) → CompletedProcess[str]

Helper to run commands with consistent behavior.

Parameters

- **cmd** (*list[str]*) – Command to execute as a list of strings
- **cwd** (*Path | None*) – Working directory for command execution, defaults to None

Returns
Completed process result

Return type
subprocess.CompletedProcess[str]

mafw.devtools.documentation.builder.**DEFAULT_MIN_TAG_REGEX = '^v([1-9][0-9]*)\\.[0-9]+\\.[0-9]+(\\.[0-9]+)?$'**

Regular expression to match stable version tags.

mafw.devtools.documentation.builder.**DOCS_SUBPATH = PosixPath('docs/source')**

The files/directories under each worktree where docs live.

mafw.devtools.documentation.builder.**OLD_VERSION_TO_BE_PATCHED = ['v1.0.0', 'v1.1.0', 'v1.2.0', 'v1.3.0', 'v1.4.0']**

Tags that require patching with the latest conf.py.

mafw.devtools.documentation.builder.**SPHINX_BUILD_CMD = 'sphinx-build'**

Sphinx build command name.

mafw.devtools.documentation.requirements

Requirements documentation constants and generators for MAFw.

This module holds the shared constants used by both the `multiversion-doc` CLI and other development tools (e.g. dependency freeze, release workflow), as well as the RST generation functions for dependency tables and Python version substitutions.

Module Attributes

REQUIREMENTS_GROUPS	Dependency groups to generate requirements documentation for.
PYTHON_VERSIONS_REQUIREMENTS_FILENAME	Filename for the generated Python substitution file.

Functions

generate_python_versions_rst(*[, repo_root])	Generate the RST substitution file for the supported Python range.
generate_requirements_rst([group_name, ...])	Generate an RST file with the dependencies of a given group.

mafw.devtools.documentation.requirements.**_load_default_python_version**(*repo_root: Path | None = None*) → str

Load the default Python version declared in `pyproject.toml`.

Reads `tool.mafw.default-python` from pyproject.toml. If the key is absent, the latest (highest) version from `tool.mafw.supported-python` is used as fallback. If the declared default version is not among the supported versions, a warning is emitted and the latest supported version is returned instead.

The result is cached via `functools.cache()` to avoid repeated TOML parsing across multiple call sites.

Parameters
repo_root (*Path | None*) – Repository root directory, defaults to auto-detection.

Returns
The default Python version as a dotted string (e.g. `"3.14"`).

Return type
str

Raises
DevtoolsError – If `tool.mafw.supported-python` cannot be loaded.

mafw.devtools.documentation.requirements.**_load_supported_python_versions**(*, *repo_root: Path | None = None*) → list[str]

Load the supported Python versions declared in `pyproject.toml`.

Parameters
repo_root (*Path | None*) – Repository root directory, defaults to auto-detection

Returns
Sorted list of supported `major.minor` versions.

Return type
list[str]

mafw.devtools.documentation.requirements.**generate_python_versions_rst**(*, *repo_root: Path | None = None*) → None

Generate the RST substitution file for the supported Python range.

The file is emitted under `docs/source/requirements` so it can be included by the general documentation and copied into the README update block.

Parameters
repo_root (*Path | None*) – Repository root directory, defaults to auto-detection

mafw.devtools.documentation.requirements.**generate_requirements_rst**(*group_name: str = 'base'*, *, *repo_root: Path | None = None*) → None

Generate an RST file with the dependencies of a given group.

The function parses pyproject.toml to retrieve the dependencies and their descriptions from the tool.mafw.dependency-description section. The generated file is saved as <group_name>_requirements.rst in the docs/source directory.

Parameters

- **group_name** (*str*) – The name of the dependency group (e.g., 'base', 'seaborn'), defaults to 'base'
- **repo_root** (*Path | None*) – Repository root directory, defaults to auto-detection

mafw.devtools.documentation.requirements.**PYTHON_VERSIONS_REQUIREMENTS_FILENAME = 'python_versions.rst'**

Filename for the generated Python substitution file.

mafw.devtools.documentation.requirements.**REQUIREMENTS_GROUPS = ['base', 'seaborn', 'devtools']**

Dependency groups to generate requirements documentation for.

mafw.devtools.documentation.server

Local documentation server management for MAFw.

This module provides the *ServerStatusEnum*, helper functions, and orchestration logic for managing a `python -m http.server` subprocess used to serve locally built versioned documentation.

Module Attributes

SERVER_DEFAULT_DIRECTORY	Default directory served by the local documentation server.

continues on next page

Table 14.54 – continued from previous page

SERVER_METADATA_FILENAME	Filename for server metadata stored inside the served directory.

Functions

format_started_at_utc(now)	Format a timestamp for server metadata.
get_server_status(directory)	Determine the current server status from metadata and process/HTTP checks.
http_probe(address, port[, timeout_s])	Probe whether the server responds at the given address:port.
process_matches_http_server(proc, address, ...)	Check whether a process resembles the expected http.server invocation.
read_server_metadata(directory)	Read local documentation server metadata if available.
server_default_directory()	Return the default directory served by the local documentation server.
server_metadata_path(directory)	Return the metadata path associated to a served directory.
server_start_impl(address, port, directory, ...)	Implementation for starting a local documentation server.
server_stop_by_metadata(directory, ...)	Stop the server based on metadata found under a served directory.
write_server_metadata(directory, metadata)	Write local documentation server metadata.

Classes

ServerStatusEnum(*values)	Possible status values for the multiversion-doc local documentation server.

class mafw.devtools.documentation.server.**ServerStatusEnum**(**values*)

Bases: StrEnum

Possible status values for the multiversion-doc local documentation server.

Variables

- **running** – The server process exists, matches the expected signature and responds over HTTP.
- **stale** – The server process exists and matches the expected signature, but does not respond over HTTP.
- **stopped** – The recorded PID does not exist or does not match the expected signature.
- **unknown** – The metadata file is missing, so no status can be determined.

static _generate_next_value_(*name*, *start*, *count*, *last_values*)

Return the lower-cased version of the member name.

mafw.devtools.documentation.server.**format_started_at_utc**(*now: datetime*) → str

Format a timestamp for server metadata.

Parameters

now (*datetime*) – A datetime instance (timezone-aware is preferred)

Returns

UTC ISO-8601 timestamp with Z suffix

Return type

str

mafw.devtools.documentation.server.**get_server_status**(*directory: Path*) → tuple[*ServerStatusEnum*, dict[str, Any] | None]

Determine the current server status from metadata and process/HTTP checks.

mafw.devtools.documentation.server.**http_probe**(*address: str*, *port: int*, *timeout_s: float = 1.0*) → bool

Probe whether the server responds at the given address:port.

Parameters

- **address** (*str*) – Server address
- **port** (*int*) – Server port
- **timeout_s** (*float*) – Timeout in seconds

Returns
True if a GET to the server root returns HTTP 200

Return type
bool

mafw.devtools.documentation.server.**process_matches_http_server**(*proc: Any*, *address: str*, *port: int*, *directory: Path*) → bool

Check whether a process resembles the expected http.server invocation.

Parameters

- **proc** (*Any*) – psutil Process instance
- **address** (*str*) – Expected bind address
- **port** (*int*) – Expected port
- **directory** (*Path*) – Expected served directory

Returns
True when the process signature matches

Return type
bool

mafw.devtools.documentation.server.**read_server_metadata**(*directory: Path*) → dict[str, Any] | None

Read local documentation server metadata if available.

Parameters
directory (*Path*) – Served directory to locate metadata under

Returns
Parsed metadata, or None when the metadata file is missing

Return type
dict[str, Any] | None

Raises
DevtoolsError – If the metadata file exists but cannot be parsed

mafw.devtools.documentation.server.**server_default_directory**() → Path

Return the default directory served by the local documentation server.

Returns
Default documentation build directory

Return type
Path

mafw.devtools.documentation.server.**server_metadata_path**(*directory: Path*) → Path

Return the metadata path associated to a served directory.

Parameters
directory (*Path*) – Served directory

Returns
Metadata JSON file path

Return type
Path

mafw.devtools.documentation.server.**server_start_impl**(*address: str*, *port: int*, *directory: Path*, *log_file: Path*, *force_restart: bool*) → None

Implementation for starting a local documentation server.

mafw.devtools.documentation.server.**server_stop_by_metadata**(*directory: Path*, *require_metadata: bool*) → None

Stop the server based on metadata found under a served directory.

mafw.devtools.documentation.server.**write_server_metadata**(*directory: Path*, *metadata: dict[str, Any]*) → None

Write local documentation server metadata.

Parameters

- **directory** (*Path*) – Served directory where metadata is stored
- **metadata** (*dict[str, Any]*) – Metadata dictionary to serialize as JSON

Raises
DevtoolsError – If the metadata file cannot be written

mafw.devtools.documentation.server.**SERVER_DEFAULT_DIRECTORY = PosixPath('docs/build')**

Default directory served by the local documentation server.

mafw.devtools.documentation.server.**SERVER_METADATA_FILENAME = '.multiversion-doc-server.json'**

Filename for server metadata stored inside the served directory.

mafw.devtools.documentation.versions

Version management helpers for MAFw versioned documentation.

This module provides functions for writing `versions.json`, creating redirect pages, mirroring version directories, pruning old versions, and generating landing pages.

Functions

ensure_versions_json_exists(outdir)	Ensure versions.json exists in outdir.
format_size(bytes_size)	Format bytes to human-readable size.
get_directory_size(path)	Calculate total size of a directory in bytes.
mirror_version(outdir, src_tag, target_tag)	Mirror a version directory from one tag to another.
prune_old_versions(outdir[, max_size_mb, ...])	Remove oldest version directories until total size is below threshold.
regenerate_versions_json_after_pruning(...)	Regenerate versions.json after pruning, excluding removed versions.
write_legacy_redirect_page(outdir)	Create a legacy redirect page at the root of the output directory.
write_redirect_page(outdir, name, target_tag)	Create a redirect page for a version alias.

continues on next page

Table 14.57 – continued from previous page

write_redirects_file(outdir)	Create a _redirects file for GitLab Pages.
write_root_landing_page(build_root[, ...])	Create a landing page for the project root with links to documentation and coverage.
write_versions_json(outdir, versions)	Write versions information to a JSON file.

mafw.devtools.documentation.versions.**ensure_versions_json_exists**(*outdir: Path*) → bool

Ensure versions.json exists in outdir. If not, try to copy from another version.

Parameters
outdir (*Path*) – Output directory that should contain versions.json

Returns
True if versions.json exists or was successfully copied

Return type
bool

mafw.devtools.documentation.versions.**format_size**(*bytes_size: float*) → str

Format bytes to human-readable size.

Parameters
bytes_size (*int*) – Size in bytes

Returns
Formatted size string

Return type
str

mafw.devtools.documentation.versions.**get_directory_size**(*path: Path*) → int

Calculate total size of a directory in bytes.

Parameters
path (*Path*) – Directory path

Returns
Total size in bytes

Return type
int

mafw.devtools.documentation.versions.**mirror_version**(*outdir: Path*, *src_tag: str*, *target_tag: str*, *use_symlink: bool = True*) → None

Mirror a version directory from one tag to another. Can use symlinks for efficiency or copy for compatibility.

Parameters

- **outdir** (*Path*) – Output directory containing version directories
- **src_tag** (*str*) – Source tag directory name
- **target_tag** (*str*) – Target tag directory name
- **use_symlink** (*bool*) – Whether to use symlink instead of copying, defaults to True

mafw.devtools.documentation.versions.**prune_old_versions**(*outdir: Path*, *max_size_mb: int = 100*, *dry_run: bool = False*) → tuple[list[str], int]

Remove oldest version directories until total size is below threshold. Always keeps 'stable', 'latest', and 'dev' (if present).

Parameters

- **outdir** (*Path*) – Output directory containing version directories
- **max_size_mb** (*int*) – Maximum size in megabytes

- **dry_run** (*bool*) – If True, only report what would be deleted

Returns
Tuple of (list of removed versions, final size in bytes)

Return type
tuple[list[str], int]

mafw.devtools.documentation.versions.**regenerate_versions_json_after_pruning**(*outdir: Path*, *removed_versions: list[str]*) → None

Regenerate versions.json after pruning, excluding removed versions.

Parameters

- **outdir** (*Path*) – Output directory containing version directories
- **removed_versions** (*list[str]*) – List of version names that were removed

mafw.devtools.documentation.versions.**write_legacy_redirect_page**(*outdir: Path*) → None

Create a legacy redirect page at the root of the output directory.

Parameters
outdir (*Path*) – Output directory for the redirect page

mafw.devtools.documentation.versions.**write_redirect_page**(*outdir: Path*, *name: str*, *target_tag: str*) → None

Create a redirect page for a version alias.

Parameters

- **outdir** (*Path*) – Output directory for the redirect page
- **name** (*str*) – Name of the redirect alias (e.g., 'stable', 'dev')
- **target_tag** (*str*) – Tag that the redirect should point to

mafw.devtools.documentation.versions.**write_redirects_file**(*outdir: Path*) → None

Create a _redirects file for GitLab Pages.

Parameters
outdir (*Path*) – Output directory for the redirects file

mafw.devtools.documentation.versions.**write_root_landing_page**(*build_root: Path*, *project_name: str = 'MAFw'*) → None

Create a landing page for the project root with links to documentation and coverage.

Parameters

- **build_root** (*Path*) – Root build directory (should contain 'doc' subdirectory)
- **project_name** (*str*) – Project name for the page title

mafw.devtools.documentation.versions.**write_versions_json**(*outdir: Path*, *versions: list[dict[str, str]]*) → None

Write versions information to a JSON file.

Parameters

- **outdir** (*Path*) – Output directory for the JSON file
- **versions** (*list[dict[str, str]]*) – List of version information dictionaries

mafw.devtools.git

Git operations for MAFw development tools.

This module centralizes all functions that shell out to `git` commands, used by the release workflow, documentation builder, and dependency management.

Module Attributes

PYPROJECT_FILE	Path to the TOML file containing the project dependencies.
STABLE_TAG_PATTERN	Regular expression used to identify stable git tags in the form `vX.Y.Z`.

Functions

check_main_branch()	Ensure the release process is executed from the `main` branch.
commit_changes(version, dry_run, *[, ...])	Commit tracked release artifacts for the target version.
commit_dependency_unfreeze(dry_run)	Commit the unfreezing of dependency upper bounds to `main`.
create_tag(version, dry_run)	Create the local git tag for the target version.
ensure_clean_git()	Ensure the git working tree is clean (excluding untracked files).
get_current_branch()	Get the name of the currently checked out branch.
get_git_tags([min_version])	Return list of (Version, tag) tuples sorted ascending.
get_last_stable_tag()	Return the latest stable tag matching `vX.Y.Z`.
git_rev_of(ref)	Get the git revision hash for a given reference.
is_ancestor(a, b)	Return True if commit a is ancestor of commit b (git merge-base --is-ancestor).
prevent_duplicate_tag(version)	Ensure the target release tag does not already exist.
push_changes(dry_run)	Push commits and tags to the remote repository.
sort_tags_semver(tags)	Sort tags using semantic versioning comparison.

mafw.devtools.git.**check_main_branch**() → None

Ensure the release process is executed from the `main` branch.

Raises

DevtoolsError – If the current branch is not `main`.

mafw.devtools.git.**commit_changes**(*version: str*, *dry_run: bool*, *, *include_changelog: bool = True*) → None

Commit tracked release artifacts for the target version.

Parameters

- **version** (*str*) – Target release version.
- **dry_run** (*bool*) – Whether command execution is disabled.
- **include_changelog** (*bool*) – Whether `CHANGELOG.md` should be staged and committed as part of the release artifacts.

mafw.devtools.git.**commit_dependency_unfreeze**(*dry_run: bool*) → None

Commit the unfreezing of dependency upper bounds to `main`.

Parameters

dry_run (*bool*) – Whether command execution is disabled.

mafw.devtools.git.**create_tag**(*version: str*, *dry_run: bool*) → str

Create the local git tag for the target version.

Parameters

- **version** (*str*) – Target release version.
- **dry_run** (*bool*) – Whether command execution is disabled.

Returns
Created tag name.

Return type
str

mafw.devtools.git.**ensure_clean_git**() → None

Ensure the git working tree is clean (excluding untracked files).

Raises
DevtoolsError – If tracked changes are present.

mafw.devtools.git.**get_current_branch**() → str

Get the name of the currently checked out branch.

Returns
Name of the current branch

Return type
str

mafw.devtools.git.**get_git_tags**(*min_version: str | None = None*) → list[tuple[Version, Any]]

Return list of (Version, tag) tuples sorted ascending.

Parameters
min_version (*str | None*) – Minimum version to consider, defaults to None

Returns
List of (Version, tag) tuples

Return type
list[tuple[Version, Any]]

mafw.devtools.git.**get_last_stable_tag**() → str | None

Return the latest stable tag matching `vX.Y.Z`.

Returns
Latest stable tag or `None` when no stable tags are present.

Return type
str | None

mafw.devtools.git.**git_rev_of**(*ref: str*) → str

Get the git revision hash for a given reference.

Parameters
ref (*str*) – Git reference (tag, branch, commit hash)

Returns
Git revision hash

Return type
str

Raises
RuntimeError – If git rev-list fails

mafw.devtools.git.**is_ancestor**(*a: str*, *b: str*) → bool

Return True if commit a is ancestor of commit b (git merge-base –is-ancestor).

Parameters

- **a** (*str*) – First commit reference
- **b** (*str*) – Second commit reference

Returns
True if a is ancestor of b

Return type
bool

mafw.devtools.git.**prevent_duplicate_tag**(*version: str*) → None

Ensure the target release tag does not already exist.

Parameters
version (*str*) – Target release version.

Raises
DevtoolsError – If v<version> already exists.

mafw.devtools.git.**push_changes**(*dry_run: bool*) → None

Push commits and tags to the remote repository.

Parameters
dry_run (*bool*) – Whether command execution is disabled.

mafw.devtools.git.**sort_tags_semver**(*tags: list[str]*) → list[str]

Sort tags using semantic versioning comparison.

Parameters
tags (*list[str]*) – List of tag strings to sort

Returns
Sorted list of tag strings

Return type
list[str]

mafw.devtools.git.**PYPROJECT_FILE = PosixPath('pyproject.toml')**

Path to the TOML file containing the project dependencies.

mafw.devtools.git.**STABLE_TAG_PATTERN = re.compile('^v(?P<major>\\d+)\\.(?P<minor>\\d+)\\.(?P<micro>\\d+)$')**

Regular expression used to identify stable git tags in the form vX.Y.Z.

mafw.devtools.gitlab

GitLab integration helpers for MAFw development tools.

This package provides HTTP helpers for communicating with the GitLab API and the Generic Package Registry.

class mafw.devtools.gitlab.**GitlabAPIConfiguration**(*api_url: str*, *on_ci: bool*, *project_id: int*, *token: str*, *token_type: Literal['job_token', 'api_token']*)

Bases: object

Configuration needed to communicate with the GitLab API.

Parameters

- **api_url** (*str*) – Base GitLab API v4 URL.
- **on_ci** (*bool*) – Whether the process runs on GitLab CI.

- **project_id** (*int*) – GitLab project numeric ID.
- **token** (*str*) – Authentication token value.
- **token_type** (*Literal['job_token', 'api_token']*) – Token kind used for authentication.

mafw.devtools.gitlab.**build_gitlab_api_configuration**(*api_url: str | None*, *project_id: int | None*, *token: str | None*) → *GitlabAPIConfiguration*

Build a GitLab API configuration from provided values and environment context.

Parameters

- **api_url** (*str | None*) – Optional override for the GitLab API URL.
- **project_id** (*int | None*) – Optional override for the GitLab project ID.
- **token** (*str | None*) – Optional override for the authentication token.

Returns
The validated GitLab API configuration.

Return type
gitlab.GitlabAPIConfiguration

Raises
ValueError – If any required configuration value is missing.

mafw.devtools.gitlab.**build_gitlab_auth_headers**(*api_config: GitlabAPIConfiguration*) → MutableMapping[str, str | bytes]

Build authentication headers for GitLab API requests.

Parameters
api_config (gitlab.GitlabAPIConfiguration) – The GitLab API configuration.

Returns
A mutable mapping of HTTP headers.

Return type
MutableMapping[str, str | bytes]

mafw.devtools.gitlab.**build_pylock_compiled_filename**(*python_version: str*) → str

Build the compiled lock filename for a given Python version.

Parameters
python_version (*str*) – Python version in `major.minor` form (e.g. `3.14`).

Returns
The compiled lock filename (e.g. `pylock.py3_14.toml`).

Return type
str

mafw.devtools.gitlab.**build_pylock_reference_filename**(*python_version: str*) → str

Build the reference lock filename for a given Python version.

Parameters
python_version (*str*) – Python version in `major.minor` form (e.g. `3.14`).

Returns
The reference lock filename (e.g. `pylock.py3_14_ref.toml`).

Return type
str

mafw.devtools.gitlab.**delete_generic_package_file**(*api_config: GitlabAPIConfiguration*, *package_id: int*, *file_id: int*, ***, *timeout_s: float = 60.0*) → bool

Delete a single file from a package version.

Parameters

- **api_config** (gitlab.GitlabAPIConfiguration) – The GitLab API configuration.
- **package_id** (*int*) – The numeric ID of the package version.
- **file_id** (*int*) – The numeric ID of the package file to delete.
- **timeout_s** (*float*) – Request timeout in seconds.

Returns
True if deleted, False if not found or forbidden.

Return type
bool

Raises
RuntimeError – If the API request fails.

mafw.devtools.gitlab.**delete_generic_package_version**(*api_config: GitlabAPIConfiguration*, *package_id: int*, *package_name: str*, *package_version: str*) → bool

Delete a package version from the GitLab Package Registry.

Parameters

- **api_config** (gitlab.GitlabAPIConfiguration) – The GitLab API configuration.
- **package_id** (*int*) – The numeric ID of the package version.
- **package_name** (*str*) – The name of the package (for error reporting).
- **package_version** (*str*) – The version of the package (for error reporting).

Returns
True if deleted, False if not found or forbidden.

Return type
bool

Raises
RuntimeError – If the API request fails.

mafw.devtools.gitlab.**download_generic_file**(*api_config: GitlabAPIConfiguration*, *package_name: str*, *package_version: str*, *file_name: str*, *download_dir: Path*, ***, *timeout_s: float = 60.0*) → Path | None

Download a file from the GitLab Generic Package Registry.

Parameters

- **api_config** (gitlab.GitlabAPIConfiguration) – The GitLab API configuration.
- **package_name** (*str*) – The name of the generic package.
- **package_version** (*str*) – The version of the generic package.
- **file_name** (*str*) – The name of the file to download.
- **download_dir** (*Path*) – The local directory where to save the file.
- **timeout_s** (*float*) – Request timeout in seconds.

Returns
The Path to the downloaded file, or None if not found.

Return type
Path | None

Raises
RuntimeError – If the API request fails.

mafw.devtools.gitlab.**iter_local_pylock_reference_files**(*directory: Path*) → list[tuple[str, Path]]

List local reference dependency files in a directory.

Parameters
directory (*Path*) – The directory to search for reference files.

Returns
A list of (python_version, file_path) tuples, sorted by version.

Return type
list[tuple[str, Path]]

mafw.devtools.gitlab.**list_generic_package_files**(*api_config: GitlabAPIConfiguration*, *package_id: int*, *, *timeout_s: float = 60.0*) → list[dict[str, Any]]

List files belonging to a specific package version.

Parameters

- **api_config** (gitlab.GitlabAPIConfiguration) – The GitLab API configuration.
- **package_id** (*int*) – The numeric ID of the package version.
- **timeout_s** (*float*) – Request timeout in seconds.

Returns
A list of file metadata dictionaries.

Return type
list[dict[str, Any]]

Raises
RuntimeError – If the API request fails.

mafw.devtools.gitlab.**list_generic_packages**(*api_config: GitlabAPIConfiguration*, *package_name: str*) → list[dict[str, Any]]

List generic packages for a given name in the GitLab Package Registry.

Parameters

- **api_config** (gitlab.GitlabAPIConfiguration) – The GitLab API configuration.
- **package_name** (*str*) – The name of the package to list.

Returns
A list of package dictionaries from the API.

Return type
list[dict[str, Any]]

Raises
RuntimeError – If the API request fails.

mafw.devtools.gitlab.**normalize_dependency_registry_item**(*item: str*) → tuple[str, str]

Normalize a registry item into a Python version and dependency file name.

Parameters
item (*str*) – A filename or a version string.

Returns
A tuple of (python_version, reference_filename).

Return type
tuple[str, str]

Raises
ValueError – If the item is invalid or unsupported.

mafw.devtools.gitlab.**normalize_mafw_version**(*version_text: str*) → str

Normalize a MAFw version override and strip an optional leading `v`.

Parameters
version_text (*str*) – The version string to normalize.

Returns
The normalized version string.

Return type
str

Raises
ValueError – If the version string is invalid.

mafw.devtools.gitlab.**parse_pylock_reference_filename**(*file_name: str*) → tuple[str, str] | None

Parse a reference dependency filename into the Python version and normalized file name.

Parameters
file_name (*str*) – The filename to parse.

Returns
A tuple of (python_version, filename) if matched, else None.

Return type
tuple[str, str] | None

mafw.devtools.gitlab.**pylock_version_tag**(*python_version: str*) → str

Return the filename-safe version tag for a Python `major.minor` version.

`uv` (>= 0.12) only accepts lock files matching `^pylock\.([^.]+\.)?toml$`, which forbids an extra dot in the filename. To comply, the dot in the Python version is replaced with an underscore (e.g. `3.14` becomes `3_14`).

Parameters
python_version (*str*) – Python version in `major.minor` form (e.g. `3.14`).

Returns
The version tag with the dot replaced by an underscore (e.g. `3_14`).

Return type
str

mafw.devtools.gitlab.**resolve_package_ids_by_version**(*api_config: GitlabAPIConfiguration*, *package_name: str*) → dict[str, int]

Resolve package IDs for a package name by version.

Parameters
- **api_config** (gitlab.GitlabAPIConfiguration) – The GitLab API configuration.
- **package_name** (*str*) – The name of the package.

Returns
A mapping of version strings to package IDs.

Return type
dict[str, int]

mafw.devtools.gitlab.**upload_generic_file**(*api_config: GitlabAPIConfiguration*, *package_name: str*, *package_version: str*, *file_path: Path*, ***, *replace_existing: bool = False*, *package_id: int | None = None*, *timeout_s: float = 60.0*) → bool

Upload a file to the GitLab Generic Package Registry.

Parameters

- **api_config** (gitlab.GitlabAPIConfiguration) – The GitLab API configuration.
- **package_name** (*str*) – The name of the generic package.
- **package_version** (*str*) – The version of the generic package.
- **file_path** (*Path*) – The local path to the file to upload.
- **replace_existing** (*bool*) – Whether to overwrite if the file already exists.
- **package_id** (*int | None*) – Optional numeric package ID to avoid extra API calls.
- **timeout_s** (*float*) – Request timeout in seconds.

Returns
True if the file was uploaded, False if skipped.

Return type
bool

Raises

- **FileNotFoundError** – If the file_path does not exist.
- **RuntimeError** – If the API request fails.

Modules

api	GitLab API configuration and authentication helpers.
docs_registry	Documentation-specific GitLab Generic Package Registry operations.
registry	GitLab Generic Package Registry operations.

mafw.devtools.gitlab.api

GitLab API configuration and authentication helpers.

author
Bulgheroni Antonio (antonio.bulgheroni@ec.europa.eu)

Functions

build_gitlab_api_configuration(api_url, ...)	Build a GitLab API configuration from provided values and environment context.
build_gitlab_auth_headers(api_config)	Build authentication headers for GitLab API requests.

Classes

GitlabAPIConfiguration(api_url, on_ci, ...)	Configuration needed to communicate with the GitLab API.

class mafw.devtools.gitlab.api.**GitlabAPIConfiguration**(*api_url: str*, *on_ci: bool*, *project_id: int*, *token: str*, *token_type: Literal['job_token', 'api_token']*)

Bases: object

Configuration needed to communicate with the GitLab API.

Parameters

- **api_url** (*str*) – Base GitLab API v4 URL.
- **on_ci** (*bool*) – Whether the process runs on GitLab CI.
- **project_id** (*int*) – GitLab project numeric ID.
- **token** (*str*) – Authentication token value.
- **token_type** (*Literal['job_token', 'api_token']*) – Token kind used for authentication.

mafw.devtools.gitlab.api.**build_gitlab_api_configuration**(*api_url: str | None*, *project_id: int | None*, *token: str | None*) → *GitlabAPIConfiguration*

Build a GitLab API configuration from provided values and environment context.

Parameters

- **api_url** (*str | None*) – Optional override for the GitLab API URL.
- **project_id** (*int | None*) – Optional override for the GitLab project ID.
- **token** (*str | None*) – Optional override for the authentication token.

Returns
The validated GitLab API configuration.

Return type
gitlab.GitlabAPIConfiguration

Raises
ValueError – If any required configuration value is missing.

mafw.devtools.gitlab.api.**build_gitlab_auth_headers**(*api_config: GitlabAPIConfiguration*) → MutableMapping[str, str | bytes]

Build authentication headers for GitLab API requests.

Parameters
api_config (gitlab.GitlabAPIConfiguration) – The GitLab API configuration.

Returns
A mutable mapping of HTTP headers.

Return type
MutableMapping[str, str | bytes]

mafw.devtools.gitlab.docs_registry

Documentation-specific GitLab Generic Package Registry operations.

This module wraps the generic GitLab registry helpers with MAFw-docs-specific defaults (package name, filename conventions).

Functions

download_docs_zip_from_gitlab_generic_regi	Download a documentation zip archive from the GitLab Generic Package Registry.
list_mafw_docs_generic_packages(api_config)	List generic packages for mafw-docs in the GitLab Package Registry.
resolve_mafw_docs_package_ids_by_version(...	Resolve package IDs for mafw-docs generic packages by version.

continues on next page

Table 14.63 – continued from previous page

upload_docs_zip_to_gitlab_generic_registry	Upload a documentation zip archive to the GitLab Generic Package Registry.

mafw.devtools.gitlab.docs_registry.**download_docs_zip_from_gitlab_generic_registry**(*api_config: GitlabAPIConfiguration*, *package_version: str*, *download_dir: Path*, *package_name: str = 'mafw-docs'*, *file_name: str | None = None*, *timeout_s: float = 60.0*) → Path | None

Download a documentation zip archive from the GitLab Generic Package Registry.

The equivalent GitLab API endpoint is:

`GET /projects/:id/packages/generic/:package_name/:package_version/:file_name`

The function first issues a HEAD request to determine if the file exists: - 404: the package file does not exist (cache miss) and `None` is returned. - 200: the file exists and is downloaded. - otherwise: a warning is printed and `None` is returned.

Authentication headers: - `JOB-TOKEN` when running on CI (`api_config.on_ci` is True) - `PRIVATE-TOKEN` for local execution

Parameters

- **api_config** (*gitlab.GitlabAPIConfiguration*) – GitLab API configuration
- **package_version** (*str*) – Package version (typically the git tag, e.g. `v2.1.0`)
- **download_dir** (*Path*) – Directory where the downloaded zip is stored
- **package_name** (*str*) – Generic package name, defaults to `mafw-docs`
- **file_name** (*str | None*) – File name to retrieve, defaults to `mafw-docs-<version>.zip`
- **timeout_s** (*float*) – Request timeout in seconds, defaults to 60.0

Returns
Path to the downloaded zip archive, or None if it does not exist or cannot be retrieved

Return type
Path | None

mafw.devtools.gitlab.docs_registry.**list_mafw_docs_generic_packages**(*api_config: GitlabAPIConfiguration*, *package_name: str = 'mafw-docs'*) → list[dict[str, Any]]

List generic packages for mafw-docs in the GitLab Package Registry.

Uses the Packages API: `GET /projects/:id/packages` and filters for `package_type=generic` and exact `name == package_name`.

Parameters

- **api_config** (gitlab.GitlabAPIConfiguration) – GitLab API configuration
- **package_name** (*str*) – Package name, defaults to `mafw-docs`

Returns
List of package dictionaries from the API

Return type
list[dict[str, Any]]

mafw.devtools.gitlab.docs_registry.**resolve_mafw_docs_package_ids_by_version**(*api_config: GitlabAPIConfiguration*, *package_name: str = 'mafw-docs'*) → dict[str, int]

Resolve package IDs for mafw-docs generic packages by version.

Parameters

- **api_config** (gitlab.GitlabAPIConfiguration) – GitLab API configuration
- **package_name** (*str*) – Package name, defaults to `mafw-docs`

Returns
Mapping from version string to package id

Return type
dict[str, int]

mafw.devtools.gitlab.docs_registry.**upload_docs_zip_to_gitlab_generic_registry**(*api_config: GitlabAPIConfiguration*, *package_version: str*, *zip_path: Path*, *package_name: str = 'mafw-docs'*, *timeout_s: float = 60.0*) → bool

Upload a documentation zip archive to the GitLab Generic Package Registry.

The equivalent GitLab API endpoint is:

`PUT /projects/:id/packages/generic/:package_name/:package_version/:file_name`

Authentication headers: - `JOB-TOKEN` when running on CI (`api_config.on_ci` is True) - `PRIVATE-TOKEN` for local execution

Parameters

- **api_config** (gitlab.GitlabAPIConfiguration) – GitLab API configuration
- **package_version** (*str*) – Package version (typically the git tag, e.g. `v2.1.0`)
- **zip_path** (*Path*) – Path to the zip file to upload
- **package_name** (*str*) – Generic package name, defaults to `mafw-docs`
- **timeout_s** (*float*) – Request timeout in seconds, defaults to 60.0

Returns
True if the file was uploaded, False if the upload was skipped because the target already exists

Return type
bool

Raises

- **FileNotFoundError** – If `zip_path` does not exist
- **RuntimeError** – If the upload fails (non-2xx response)

mafw.devtools.gitlab.registry

GitLab Generic Package Registry operations.

This module centralizes the HTTP helpers for uploading, downloading, listing, and deleting files in the GitLab Generic Package Registry.

author
Bulgheroni Antonio (antonio.bulgheroni@ec.europa.eu)

Functions

build_pylock_compiled_filename(python_versior	Build the compiled lock filename for a given Python version.
build_pylock_reference_filename(python_versi	Build the reference lock filename for a given Python version.
delete_generic_package_file(api_config, ...)	Delete a single file from a package version.
delete_generic_package_version(api_config, ...)	Delete a package version from the GitLab Package Registry.
download_generic_file(api_config, ...[, ...])	Download a file from the GitLab Generic Package Registry.
iter_local_pylock_reference_files(directory)	List local reference dependency files in a directory.
list_generic_package_files(api_config, ...)	List files belonging to a specific package version.
list_generic_packages(api_config, package_name)	List generic packages for a given name in the GitLab Package Registry.
normalize_dependency_registry_item(item)	Normalize a registry item into a Python version and dependency file name.
normalize_mafw_version(version_text)	Normalize a MAFw version override and strip an optional leading `v`.
parse_pylock_reference_filename(file_name)	Parse a reference dependency filename into the Python version and normalized file name.
pylock_version_tag(python_version)	Return the filename-safe version tag for a Python `major.minor` version.
resolve_package_ids_by_version(api_config, ...)	Resolve package IDs for a package name by version.
upload_generic_file(api_config, ...[, ...])	Upload a file to the GitLab Generic Package Registry.

mafw.devtools.gitlab.registry.**_infer_content_type**(*file_path: Path*) → str

Infer a Content-Type from the file extension.

Parameters
file_path (*Path*) – Path to the file.

Returns
A MIME type string.

Return type
str

mafw.devtools.gitlab.registry.**build_pylock_compiled_filename**(*python_version: str*) → str

Build the compiled lock filename for a given Python version.

Parameters
python_version (*str*) – Python version in `major.minor` form (e.g. `3.14`).

Returns
The compiled lock filename (e.g. `pylock.py3_14.toml`).

Return type
str

mafw.devtools.gitlab.registry.**build_pylock_reference_filename**(*python_version: str*) → str

Build the reference lock filename for a given Python version.

Parameters
python_version (*str*) – Python version in `major.minor` form (e.g. `3.14`).

Returns
The reference lock filename (e.g. `pylock.py3_14_ref.toml`).

Return type
str

mafw.devtools.gitlab.registry.**delete_generic_package_file**(*api_config: GitlabAPIConfiguration*, *package_id: int*, *file_id: int*, ***, *timeout_s: float = 60.0*) → bool

Delete a single file from a package version.

Parameters

- **api_config** (gitlab.GitlabAPIConfiguration) – The GitLab API configuration.
- **package_id** (*int*) – The numeric ID of the package version.
- **file_id** (*int*) – The numeric ID of the package file to delete.
- **timeout_s** (*float*) – Request timeout in seconds.

Returns
True if deleted, False if not found or forbidden.

Return type
bool

Raises
RuntimeError – If the API request fails.

mafw.devtools.gitlab.registry.**delete_generic_package_version**(*api_config: GitlabAPIConfiguration*, *package_id: int*, *package_name: str*, *package_version: str*) → bool

Delete a package version from the GitLab Package Registry.

Parameters

- **api_config** (gitlab.GitlabAPIConfiguration) – The GitLab API configuration.
- **package_id** (*int*) – The numeric ID of the package version.
- **package_name** (*str*) – The name of the package (for error reporting).
- **package_version** (*str*) – The version of the package (for error reporting).

Returns
True if deleted, False if not found or forbidden.

Return type
bool

Raises
RuntimeError – If the API request fails.

mafw.devtools.gitlab.registry.**download_generic_file**(*api_config: GitlabAPIConfiguration*, *package_name: str*, *package_version: str*, *file_name: str*, *download_dir: Path*, ***, *timeout_s: float = 60.0*) → Path | None

Download a file from the GitLab Generic Package Registry.

Parameters

- **api_config** (gitlab.GitlabAPIConfiguration) – The GitLab API configuration.
- **package_name** (*str*) – The name of the generic package.
- **package_version** (*str*) – The version of the generic package.
- **file_name** (*str*) – The name of the file to download.
- **download_dir** (*Path*) – The local directory where to save the file.
- **timeout_s** (*float*) – Request timeout in seconds.

Returns
The Path to the downloaded file, or None if not found.

Return type
Path | None

Raises
RuntimeError – If the API request fails.

mafw.devtools.gitlab.registry.**iter_local_pylock_reference_files**(*directory: Path*) → list[tuple[str, Path]]

List local reference dependency files in a directory.

Parameters
directory (*Path*) – The directory to search for reference files.

Returns
A list of (python_version, file_path) tuples, sorted by version.

Return type
list[tuple[str, Path]]

mafw.devtools.gitlab.registry.**list_generic_package_files**(*api_config: GitlabAPIConfiguration*, *package_id: int*, ***, *timeout_s: float = 60.0*) → list[dict[str, Any]]

List files belonging to a specific package version.

Parameters
- **api_config** (gitlab.GitlabAPIConfiguration) – The GitLab API configuration.
- **package_id** (*int*) – The numeric ID of the package version.
- **timeout_s** (*float*) – Request timeout in seconds.

Returns
A list of file metadata dictionaries.

Return type
list[dict[str, Any]]

Raises
RuntimeError – If the API request fails.

mafw.devtools.gitlab.registry.**list_generic_packages**(*api_config: GitlabAPIConfiguration*, *package_name: str*) → list[dict[str, Any]]

List generic packages for a given name in the GitLab Package Registry.

Parameters
- **api_config** (gitlab.GitlabAPIConfiguration) – The GitLab API configuration.
- **package_name** (*str*) – The name of the package to list.

Returns
A list of package dictionaries from the API.

Return type
list[dict[str, Any]]

Raises
RuntimeError – If the API request fails.

mafw.devtools.gitlab.registry.**normalize_dependency_registry_item**(*item: str*) → tuple[str, str]

Normalize a registry item into a Python version and dependency file name.

Parameters
item (*str*) – A filename or a version string.

Returns
A tuple of (python_version, reference_filename).

Return type
tuple[str, str]

Raises
ValueError – If the item is invalid or unsupported.

mafw.devtools.gitlab.registry.**normalize_mafw_version**(*version_text: str*) → str

Normalize a MAFw version override and strip an optional leading `v`.

Parameters
version_text (*str*) – The version string to normalize.

Returns
The normalized version string.

Return type
str

Raises
ValueError – If the version string is invalid.

mafw.devtools.gitlab.registry.**parse_pylock_reference_filename**(*file_name: str*) → tuple[str, str] | None

Parse a reference dependency filename into the Python version and normalized file name.

Parameters
file_name (*str*) – The filename to parse.

Returns
A tuple of (python_version, filename) if matched, else None.

Return type
tuple[str, str] | None

mafw.devtools.gitlab.registry.**pylock_version_tag**(*python_version: str*) → str

Return the filename-safe version tag for a Python `major.minor` version.

uv (>= 0.12) only accepts lock files matching `^pylock\.([^.]+\.)?toml$`, which forbids an extra dot in the filename. To comply, the dot in the Python version is replaced with an underscore (e.g. `3.14` becomes `3_14`).

Parameters
python_version (*str*) – Python version in `major.minor` form (e.g. `3.14`).

Returns
The version tag with the dot replaced by an underscore (e.g. `3_14`).

Return type
str

mafw.devtools.gitlab.registry.**resolve_package_ids_by_version**(*api_config: GitlabAPIConfiguration*, *package_name: str*) → dict[str, int]

Resolve package IDs for a package name by version.

Parameters

- **api_config** (gitlab.GitlabAPIConfiguration) – The GitLab API configuration.
- **package_name** (*str*) – The name of the package.

Returns
A mapping of version strings to package IDs.

Return type
dict[str, int]

mafw.devtools.gitlab.registry.**upload_generic_file**(*api_config: GitlabAPIConfiguration*, *package_name: str*, *package_version: str*, *file_path: Path*, ***, *replace_existing: bool = False*, *package_id: int | None = None*, *timeout_s: float = 60.0*) → bool

Upload a file to the GitLab Generic Package Registry.

Parameters

- **api_config** (*gitlab.GitlabAPIConfiguration*) – The GitLab API configuration.
- **package_name** (*str*) – The name of the generic package.
- **package_version** (*str*) – The version of the generic package.
- **file_path** (*Path*) – The local path to the file to upload.
- **replace_existing** (*bool*) – Whether to overwrite if the file already exists.
- **package_id** (*int | None*) – Optional numeric package ID to avoid extra API calls.
- **timeout_s** (*float*) – Request timeout in seconds.

Returns
True if the file was uploaded, False if skipped.

Return type
bool

Raises

- **FileNotFoundError** – If the file_path does not exist.
- **RuntimeError** – If the API request fails.

mafw.devtools.gitlab.registry.**_CONTENT_TYPE_MAP: Final[dict[str, str]] = {'.gz': 'application/gzip', '.json': 'application/json', '.tar': 'application/x-tar', '.toml': 'application/toml', '.zip': 'application/zip'}**

Mapping of file extensions to Content-Type values for registry uploads.

mafw.devtools.gitlab.registry.**_MAFW_VERSION_RE: Final[Pattern[str]] = re.compile('^(?:v)?(?P<version>\\d+\\.\\d+\\.\\d+)$')**

Pattern for accepted MAFw version overrides.

mafw.devtools.gitlab.registry.**_PYLOCK_RE: Final[Pattern[str]] = re.compile('^pylock\\.py(?P<python_version>3_\\d+)_ref\\.toml$')**

Pattern for MAFw reference dependency lock files.

The Python version uses an underscore instead of a dot (e.g. `3_14`) so that the filename matches the uv lockfile naming constraint `^pylock\.([^.]+\.)?toml$`.

mafw.devtools.release

Release management utilities for MAFw development tools.

This package provides business logic for version bumping, changelog generation, release note creation, and git workflow operations during the release process.

class mafw.devtools.release.**DeprecatedOption**(**args: Any*, *deprecated_message: str = ''*, ***kwargs: Any*)

Bases: `Option`

A Click Option subclass that emits a DeprecationWarning when explicitly used.

Use this class as `cls=DeprecatedOption` in a `@click.option` decorator to mark an option as deprecated. The warning is only emitted when the user explicitly provides the option on the command line; default-value resolution does **not** trigger the warning. The resolved value is passed through to the command callback unchanged.

Parameters
deprecated_message (*str*) – Warning text emitted when the option is explicitly provided on the command line.

Added in version 2.2.

consume_value(*ctx: Context*, *opts: Mapping[str, Parameter]*) → tuple[Any, ParameterSource]

Intercept value consumption to detect explicit CLI usage.

Parameters
- **ctx** (*click.Context*) – Current Click context.
- **opts** (*Mapping[str, click.Parameter]*) – Parsed option tokens from the command line.

Returns
Tuple of (value, source) as returned by the parent implementation.

Return type
tuple[Any, click.core.ParameterSource]

class mafw.devtools.release.**DocTargetVersionPrompt**(**args: Any*, *min_version: str | None = None*, ***kwargs: Any*)

Bases: `Prompt`

Prompt that validates documentation target versions in `major.minor` form.

Initialize the prompt with an optional minimum allowed version.

Parameters
min_version (*str | None*) – Lowest allowed documentation target version in `major.minor` form.

process_response(*value: str*) → str

Validate and normalize the documentation target version entered by the user.

Parameters
value (*str*) – Raw user input.

Returns
Normalized version string.

Return type
str

Raises
InvalidResponse – If the value does not match `major.minor` or is below the minimum.

validate_error_message = 'Please enter a version in major.minor form, for example 2.4.'

Message shown when the entered documentation target version is invalid.

mafw.devtools.release.**bump_version**(*segment: str*, ***, *dry_run: bool*) → str

Compute or execute the version bump depending on dry-run mode.

Hatch is used as single source of truth for the resolved target version. In dry-run mode, the `__about__.py` file is temporarily rewritten by Hatch and restored afterwards so the git worktree remains unchanged.

Parameters
- **segment** (*str*) – Hatch selector segments (comma-separated).

- **dry_run** (*bool*) – Whether command execution is disabled.

Returns
New version string.

Return type
str

mafw.devtools.release.**check_main_branch**() → None

Ensure the release process is executed from the `main` branch.

Raises
DevtoolsError – If the current branch is not `main`.

mafw.devtools.release.**check_missing_release**(*current_version: str*) → None

Prevent creating a new release when a stable release is already missing a tag.

Parameters
current_version (*str*) – Current project version.

Raises
DevtoolsError – If a missing stable release is detected.

mafw.devtools.release.**classify_version**(*version: str*) → Literal['stable', 'rc', 'alpha', 'beta']

Classify a version string into stable/rc/alpha/beta.

Parameters
version (*str*) – Version string to classify.

Returns
Classified version kind.

Return type
str

Raises
DevtoolsError – If the version cannot be parsed.

mafw.devtools.release.**commit_changes**(*version: str*, *dry_run: bool*, *, *include_changelog: bool = True*) → None

Commit tracked release artifacts for the target version.

Parameters

- **version** (*str*) – Target release version.
- **dry_run** (*bool*) – Whether command execution is disabled.
- **include_changelog** (*bool*) – Whether `CHANGELOG.md` should be staged and committed as part of the release artifacts.

mafw.devtools.release.**create_release_note**(*version: str*, *dry_run: bool*) → Path

Create the release note markdown file for the given version.

Change sections are copied from the generated changelog for the same version to ensure release notes stay aligned with changelog content.

Parameters

- **version** (*str*) – Target release version.
- **dry_run** (*bool*) – Whether command execution is disabled.

Returns
Path to the release note markdown file.

Return type
Path

Raises
DevtoolsError – If the template is missing.

mafw.devtools.release.**create_tag**(*version: str*, *dry_run: bool*) → str

Create the local git tag for the target version.

Parameters

- **version** (*str*) – Target release version.
- **dry_run** (*bool*) – Whether command execution is disabled.

Returns
Created tag name.

Return type
str

mafw.devtools.release.**ensure_clean_git**() → None

Ensure the git working tree is clean (excluding untracked files).

Raises
DevtoolsError – If tracked changes are present.

mafw.devtools.release.**extract_change_sections_from_changelog**(*version: str*) → dict[str, str]

Extract release change sections from `CHANGELOG.md`.

Parameters
version (*str*) – Target release version without leading `v`.

Returns
Mapping from release-note category key to markdown content.

Return type
dict[str, str]

Raises
DevtoolsError – If changelog file does not exist or the requested section is missing.

mafw.devtools.release.**extract_change_sections_from_changelog_block**(*changelog_block: str*) → dict[str, str]

Extract release change sections from a changelog block.

The parsing logic expects the block to contain level-3 headings (`###`) with standardized labels (e.g. "New Features", "Bug Fixes", ...).

Parameters
changelog_block (*str*) – Changelog markdown block to parse.

Returns
Mapping from release-note category key to markdown content.

Return type
dict[str, str]

mafw.devtools.release.**extract_version_changelog_block**(*changelog_content: str*, *version: str*) → str

Extract the changelog block associated with the target version.

Parameters

- **changelog_content** (*str*) – Complete changelog markdown text.
- **version** (*str*) – Target release version without leading `v`.

Returns
Markdown block for the selected release.

Return type
str

Raises
DevtoolsError – If no section for the target version is found.

mafw.devtools.release.**generate_changelog**(*version: str*, *dry_run: bool*) → None

Generate the project changelog for the target release.

Parameters

- **version** (*str*) – Target release version.
- **dry_run** (*bool*) – Whether command execution is disabled.

mafw.devtools.release.**get_contributors**(*since_ref: str*) → list[str]

Collect contributor names since the baseline reference.

Parameters
since_ref (*str*) – Baseline reference to compare against `HEAD`.

Returns
Ordered list of contributor names.

Return type
list[str]

mafw.devtools.release.**get_last_stable_tag**() → str | None

Return the latest stable tag matching `vX.Y.Z`.

Returns
Latest stable tag or `None` when no stable tags are present.

Return type
str | None

mafw.devtools.release.**get_release_note_base_ref**() → str

Determine the git reference used as baseline for release-note metadata.

Returns
Last stable tag if available, otherwise the first commit hash.

Return type
str

mafw.devtools.release.**get_release_statistics**(*since_ref: str*) → tuple[str, str]

Collect release statistics since the baseline reference.

Parameters
since_ref (*str*) – Baseline reference to compare against `HEAD`.

Returns
Commit count and short diff statistics.

Return type
tuple[str, str]

mafw.devtools.release.**normalize_hatch_segments**(*segments: str*) → str

Normalize the user-provided segment selector to a Hatch-compatible token list.

The selector supports comma-separated segments and follows Hatch semantics. Examples:

- `minor,rc`: bump minor and create/reset an RC suffix.
- `rc`: increment the release-candidate counter only.
- `alpha` / `beta`: create alpha/beta pre-release suffix.
- `release`: remove any pre-release suffix (stable release).

Parameters
segments (*str*) – Raw selector as passed on the command line.

Returns
Normalized Hatch selector (comma-separated, lowercase, no whitespace).

Return type
str

Raises
DevtoolsError – If the selector is empty or contains unsupported segments.

mafw.devtools.release.**parse_stable_tag**(*tag: str*) → tuple[int, int, int]

Parse a stable tag in the form `vX.Y.Z`.

Parameters
tag (*str*) – Stable tag value.

Returns
Parsed stable version tuple.

Return type
tuple[int, int, int]

Raises
DevtoolsError – If the tag does not match `vX.Y.Z`.

mafw.devtools.release.**parse_version**(*version: str*) → tuple[int, int, int, int | None]

Parse a version string in the form `X.Y.Z` or with a pre-release suffix.

Supported suffixes follow Hatch/PEP 440 conventions:

- Release candidates: `X.Y.ZrcN`
- Alpha releases: `X.Y.ZaN`
- Beta releases: `X.Y.ZbN`

Parameters
version (*str*) – Version string to parse.

Returns
Parsed major, minor, micro, and optional pre-release index.

Return type
tuple[int, int, int, int | None]

Raises
DevtoolsError – If the version format is unsupported.

mafw.devtools.release.**prevent_duplicate_tag**(*version: str*) → None

Ensure the target release tag does not already exist.

Parameters
version (*str*) – Target release version.

Raises
DevtoolsError – If v<version> already exists.

mafw.devtools.release.**push_changes**(*dry_run: bool*) → None

Push commits and tags to the remote repository.

Parameters
dry_run (*bool*) – Whether command execution is disabled.

mafw.devtools.release.**read_current_version**() → str

Read the project version using `hatch version`.

Returns
Current project version string.

Return type
str

Raises
DevtoolsError – If the version cannot be extracted.

mafw.devtools.release.**render_release_note_section**(*content: str*, *header: str*, *section_markdown: str*) → str

Replace a release-note section in the markdown template.

If the provided section markdown is empty, the entire section (header and placeholder) is removed from the content.

Parameters

- **content** (*str*) – Current release note markdown text.
- **header** (*str*) – Section header to replace.
- **section_markdown** (*str*) – Markdown content to insert under the section header.

Returns
Updated markdown text.

Return type
str

Raises
DevtoolsError – If the section header cannot be matched in template.

mafw.devtools.release.**update_doc_target_version**(*version: str*, *dry_run: bool*) → None

Update the documentation target version in `src/mafw/__about__.py`.

Parameters

- **version** (*str*) – Target documentation version in `major.minor` form.
- **dry_run** (*bool*) – Whether filesystem changes are disabled.

mafw.devtools.release.**update_notice_version**(*version: str*, *dry_run: bool*) → None

Update the version in `NOTICE.txt` to match the release version.

Parameters

- **version** (*str*) – Target release version.
- **dry_run** (*bool*) – Whether filesystem changes are disabled.

Raises
DevtoolsError – If NOTICE.txt is missing or has unexpected format.

Modules

changelog	Changelog generation and parsing utilities for MAFw releases.
checks	Release validation checks for MAFw.
notes	Release note generation utilities for MAFw releases.
versioning	Version parsing, classification, and bumping utilities for MAFw releases.

mafw.devtools.release.changelog

Changelog generation and parsing utilities for MAFw releases.

This module contains the business logic for generating the project changelog and extracting version-specific change sections from it.

Module Attributes

CHANGELOG_FILE	Path to the changelog file that is regenerated for each release.
RELEASE_SECTION_HEADERS	Release note section headers used in the markdown template.

Functions

extract_change_sections_from_changelog(vers	Extract release change sections from `CHANGELOG.md`.
extract_change_sections_from_changelog_blo	Extract release change sections from a changelog block.
extract_version_changelog_block(...)	Extract the changelog block associated with the target version.
generate_changelog(version, dry_run)	Generate the project changelog for the target release.

mafw.devtools.release.changelog.**_classify_changelog_subsection**(*subsection_heading: str*) → str | None

Map a changelog subsection heading to a release-note category key.

Parameters
subsection_heading (*str*) – Raw level-3 heading from changelog.

Returns
Category key, or `None` if the heading is not mappable.

Return type
str | None

mafw.devtools.release.changelog.**extract_change_sections_from_changelog**(*version: str*) → dict[str, str]

Extract release change sections from `CHANGELOG.md`.

Parameters
version (*str*) – Target release version without leading `v`.

Returns
Mapping from release-note category key to markdown content.

Return type
dict[str, str]

Raises
DevtoolsError – If changelog file does not exist or the requested section is missing.

mafw.devtools.release.changelog.**extract_change_sections_from_changelog_block**(*changelog_block: str*) → dict[str, str]

Extract release change sections from a changelog block.

The parsing logic expects the block to contain level-3 headings (`###`) with standardized labels (e.g. "New Features", "Bug Fixes", ...).

Parameters
changelog_block (*str*) – Changelog markdown block to parse.

Returns
Mapping from release-note category key to markdown content.

Return type
dict[str, str]

mafw.devtools.release.changelog.**extract_version_changelog_block**(*changelog_content: str*, *version: str*) → str

Extract the changelog block associated with the target version.

Parameters

- **changelog_content** (*str*) – Complete changelog markdown text.
- **version** (*str*) – Target release version without leading `v`.

Returns
Markdown block for the selected release.

Return type
str

Raises
DevtoolsError – If no section for the target version is found.

mafw.devtools.release.changelog.**generate_changelog**(*version: str*, *dry_run: bool*) → None

Generate the project changelog for the target release.

Parameters

- **version** (*str*) – Target release version.
- **dry_run** (*bool*) – Whether command execution is disabled.

mafw.devtools.release.changelog.**CHANGELOG_FILE = PosixPath('CHANGELOG.md')**

Path to the changelog file that is regenerated for each release.

mafw.devtools.release.changelog.**RELEASE_SECTION_HEADERS = {'bug_fixes': '## Bug Fixes', 'deprecated': '## Deprecated', 'new_features': '## New Features', 'other_changes': '## * Other Changes', 'refactorings': '## Refactorings', 'removed': '## Removals', 'security': '## Security'}**

Release note section headers used in the markdown template.

mafw.devtools.release.checks

Release validation checks for MAFw.

This module contains non-git business logic checks used during the release process: tag parsing and missing-release detection.

Functions

check_missing_release(current_version)	Prevent creating a new release when a stable release is already missing a tag.
parse_stable_tag(tag)	Parse a stable tag in the form `vX.Y.Z`.

mafw.devtools.release.checks.**check_missing_release**(*current_version: str*) → None

Prevent creating a new release when a stable release is already missing a tag.

Parameters
current_version (*str*) – Current project version.

Raises
DevtoolsError – If a missing stable release is detected.

mafw.devtools.release.checks.**parse_stable_tag**(*tag: str*) → tuple[int, int, int]

Parse a stable tag in the form `vX.Y.Z`.

Parameters
tag (*str*) – Stable tag value.

Returns
Parsed stable version tuple.

Return type
tuple[int, int, int]

Raises
DevtoolsError – If the tag does not match `vX.Y.Z`.

mafw.devtools.release.notes

Release note generation utilities for MAFw releases.

This module contains the business logic for creating release notes from a markdown template, collecting git statistics, and resolving contributors.

Module Attributes

RELEASE_TEMPLATE_FILE	Path to the markdown template used to build release notes.

Functions

create_release_note(version, dry_run)	Create the release note markdown file for the given version.
get_contributors(since_ref)	Collect contributor names since the baseline reference.
get_release_note_base_ref()	Determine the git reference used as baseline for release-note metadata.
get_release_statistics(since_ref)	Collect release statistics since the baseline reference.
render_release_note_section(content, header, ...)	Replace a release-note section in the markdown template.

mafw.devtools.release.notes.**create_release_note**(*version: str*, *dry_run: bool*) → Path

Create the release note markdown file for the given version.

Change sections are copied from the generated changelog for the same version to ensure release notes stay aligned with changelog content.

Parameters

- **version** (*str*) – Target release version.
- **dry_run** (*bool*) – Whether command execution is disabled.

Returns
Path to the release note markdown file.

Return type
Path

Raises
DevtoolsError – If the template is missing.

mafw.devtools.release.notes.**get_contributors**(*since_ref: str*) → list[str]

Collect contributor names since the baseline reference.

Parameters
since_ref (*str*) – Baseline reference to compare against HEAD.

Returns
Ordered list of contributor names.

Return type
list[str]

mafw.devtools.release.notes.**get_release_note_base_ref**() → str

Determine the git reference used as baseline for release-note metadata.

Returns
Last stable tag if available, otherwise the first commit hash.

Return type
str

mafw.devtools.release.notes.**get_release_statistics**(*since_ref: str*) → tuple[str, str]

Collect release statistics since the baseline reference.

Parameters
since_ref (*str*) – Baseline reference to compare against HEAD.

Returns
Commit count and short diff statistics.

Return type
tuple[str, str]

mafw.devtools.release.notes.**render_release_note_section**(*content: str*, *header: str*, *section_markdown: str*) → str

Replace a release-note section in the markdown template.

If the provided section markdown is empty, the entire section (header and placeholder) is removed from the content.

Parameters

- **content** (*str*) – Current release note markdown text.
- **header** (*str*) – Section header to replace.
- **section_markdown** (*str*) – Markdown content to insert under the section header.

Returns
Updated markdown text.

Return type
str

Raises
DevtoolsError – If the section header cannot be matched in template.

mafw.devtools.release.notes.**RELEASE_TEMPLATE_FILE = PosixPath('.gitlab/release_templates/Default.md')**

Path to the markdown template used to build release notes.

mafw.devtools.release.versioning

Version parsing, classification, and bumping utilities for MAFw releases.

This module contains the business logic for version string parsing, classification, validation, and the Hatch-based version bumping workflow.

Module Attributes

VALID_HATCH_SEGMENTS	Allowed Hatch version segments supported by this script.
VERSION_PATTERN	Regular expression used to parse local version strings (stable/alpha/beta/rc).
STABLE_TAG_PATTERN	Regular expression used to identify stable git tags in the form `vX.Y.Z`.
ABOUT_FILE	Path to the version source file managed by `hatch version`.
NOTICE_FILE	Path to the notice file containing the public project version.
NOTICE_VERSION_PATTERN	Pattern used to update the version block in `NOTICE.txt`.
DOC_TARGET_VERSION_PATTERN	Pattern used to validate documentation target version strings.
VersionKind	Supported release kinds used to drive changelog and release-note behavior.

Functions

bump_version(segment, *, dry_run)	Compute or execute the version bump depending on dry-run mode.
classify_version(version)	Classify a version string into stable/rc/alpha/beta.
compute_doc_target_version(version, segments)	Determine the new documentation target version for a release.
next_minor_version(version)	Compute the next `major.minor` target from a release version.
normalize_hatch_segments(segments)	Normalize the user-provided segment selector to a Hatch-compatible token list.
parse_version(version)	Parse a version string in the form `X.Y.Z` or with a pre-release suffix.
read_current_version()	Read the project version using `hatch version`.
update_doc_target_version(version, dry_run)	Update the documentation target version in `src/mafw/__about__.py`.
update_notice_version(version, dry_run)	Update the version in `NOTICE.txt` to match the release version.
validate_doc_target_version(version)	Validate a documentation target version string.

Classes

DocTargetVersionPrompt(*args[, min_version])	Prompt that validates documentation target versions in `major.minor` form.

class mafw.devtools.release.versioning.**DocTargetVersionPrompt**(**args: Any*, *min_version: str | None = None*, ***kwargs: Any*)

Bases: `Prompt`

Prompt that validates documentation target versions in `major.minor` form.

Initialize the prompt with an optional minimum allowed version.

Parameters

min_version (*str | None*) – Lowest allowed documentation target version in `major.minor` form.

process_response(*value: str*) → str

Validate and normalize the documentation target version entered by the user.

Parameters
value (*str*) – Raw user input.

Returns
Normalized version string.

Return type
str

Raises
InvalidResponse – If the value does not match `major.minor` or is below the minimum.

validate_error_message = 'Please enter a version in major.minor form, for example 2.4.'

Message shown when the entered documentation target version is invalid.

mafw.devtools.release.versioning.**_parse_major_minor**(*version: str*) → tuple[int, int]

Parse a version string in `major.minor` form.

Parameters
version (*str*) – Version string to parse.

Returns
Parsed major and minor components.

Return type
tuple[int, int]

Raises
DevtoolsError – If the version format is invalid.

mafw.devtools.release.versioning.**bump_version**(*segment: str*, *, *dry_run: bool*) → str

Compute or execute the version bump depending on dry-run mode.

Hatch is used as single source of truth for the resolved target version. In dry-run mode, the `__about__.py` file is temporarily rewritten by Hatch and restored afterwards so the git worktree remains unchanged.

Parameters

- **segment** (*str*) – Hatch selector segments (comma-separated).
- **dry_run** (*bool*) – Whether command execution is disabled.

Returns
New version string.

Return type
str

mafw.devtools.release.versioning.**classify_version**(*version: str*) → Literal['stable', 'rc', 'alpha', 'beta']

Classify a version string into stable/rc/alpha/beta.

Parameters
version (*str*) – Version string to classify.

Returns
Classified version kind.

Return type
str

Raises
DevtoolsError – If the version cannot be parsed.

mafw.devtools.release.versioning.**compute_doc_target_version**(*version: str*, *segments: str*, *override: str | None = None*) → str

Determine the new documentation target version for a release.

Parameters

- **version** (*str*) – Bumped release version.
- **segments** (*str*) – Normalized Hatch selector used for the release.
- **override** (*str | None*) – Optional explicit documentation target override.

Returns
Documentation target version in `major.minor` form.

Return type
str

mafw.devtools.release.versioning.**next_minor_version**(*version: str*) → str

Compute the next `major.minor` target from a release version.

Parameters
version (*str*) – Release version in `major.minor.micro` form.

Returns
Next documentation target version.

Return type
str

mafw.devtools.release.versioning.**normalize_hatch_segments**(*segments: str*) → str

Normalize the user-provided segment selector to a Hatch-compatible token list.

The selector supports comma-separated segments and follows Hatch semantics. Examples:

- `minor,rc`: bump minor and create/reset an RC suffix.
- `rc`: increment the release-candidate counter only.
- `alpha` / `beta`: create alpha/beta pre-release suffix.
- `release`: remove any pre-release suffix (stable release).

Parameters
segments (*str*) – Raw selector as passed on the command line.

Returns
Normalized Hatch selector (comma-separated, lowercase, no whitespace).

Return type
str

Raises
DevtoolsError – If the selector is empty or contains unsupported segments.

mafw.devtools.release.versioning.**parse_version**(*version: str*) → tuple[int, int, int, int | None]

Parse a version string in the form `X.Y.Z` or with a pre-release suffix.

Supported suffixes follow Hatch/PEP 440 conventions:

- Release candidates: `X.Y.ZrcN`
- Alpha releases: `X.Y.ZaN`
- Beta releases: `X.Y.ZbN`

Parameters
version (*str*) – Version string to parse.

Returns
Parsed major, minor, micro, and optional pre-release index.

Return type
tuple[int, int, int, int | None]

Raises
DevtoolsError – If the version format is unsupported.

mafw.devtools.release.versioning.**read_current_version**() → str

Read the project version using `hatch version`.

Returns
Current project version string.

Return type
str

Raises
DevtoolsError – If the version cannot be extracted.

mafw.devtools.release.versioning.**update_doc_target_version**(*version: str*, *dry_run: bool*) → None

Update the documentation target version in `src/mafw/__about__.py`.

Parameters

- **version** (*str*) – Target documentation version in `major.minor` form.
- **dry_run** (*bool*) – Whether filesystem changes are disabled.

mafw.devtools.release.versioning.**update_notice_version**(*version: str*, *dry_run: bool*) → None

Update the version in `NOTICE.txt` to match the release version.

Parameters

- **version** (*str*) – Target release version.
- **dry_run** (*bool*) – Whether filesystem changes are disabled.

Raises
DevtoolsError – If NOTICE.txt is missing or has unexpected format.

mafw.devtools.release.versioning.**validate_doc_target_version**(*version: str*) → str

Validate a documentation target version string.

Parameters
version (*str*) – Version string to validate.

Returns
Normalized version string.

Return type
str

Raises
DevtoolsError – If the version format is invalid.

mafw.devtools.release.versioning.**ABOUT_FILE = PosixPath('src/mafw/__about__.py')**

Path to the version source file managed by `hatch version`.

mafw.devtools.release.versioning.**DOC_TARGET_VERSION_PATTERN = re.compile('^\\d+\\.\\d+$')**

Pattern used to validate documentation target version strings.

mafw.devtools.release.versioning.**NOTICE_FILE = PosixPath('NOTICE.txt')**

Path to the notice file containing the public project version.

mafw.devtools.release.versioning.**NOTICE_VERSION_PATTERN = re.compile('MAFw - Modular Analysis Framework\\n\\nversion:\\s*V[0-9]+\\.[0-9]+\\.[0-9]+(?:[-a-zA-Z0-9\\.\\-_]+)?', re.MULTILINE)**

Pattern used to update the version block in `NOTICE.txt`.

mafw.devtools.release.versioning.**STABLE_TAG_PATTERN = re.compile('^v(?P<major>\\d+)\\.(?P<minor>\\d+)\\.(?P<micro>\\d+)$')**

Regular expression used to identify stable git tags in the form `vX.Y.Z`.

mafw.devtools.release.versioning.**VALID_HATCH_SEGMENTS: Final[tuple[str, ...]] = ('major', 'minor', 'micro', 'rc', 'alpha', 'beta', 'release')**

Allowed Hatch version segments supported by this script.

mafw.devtools.release.versioning.**VERSION_PATTERN = re.compile('^(?P<major>\\d+)\\.(?P<minor>\\d+)\\.(?P<micro>\\d+)(?:(?P<suffix>rc|a|b)(?P<suffix_num>\\d+))?$')**

Regular expression used to parse local version strings (stable/alpha/beta/rc).

mafw.devtools.release.versioning.**VersionKind**

Supported release kinds used to drive changelog and release-note behavior.

alias of `Literal`['stable', 'rc', 'alpha', 'beta']

mafw.devtools.toolchain

Toolchain management package for MAFw development tools.

This package provides the abstract interface, data models, helpers, and registry for centralized management of all development tools used by MAFw. Each tool implements the *ToolChainTool* contract, enabling uniform bootstrapping, version checking, updating, and configuration verification.

The `ensure_devtools_available()` guard is performed at the CLI entry-point level rather than at package import time, allowing individual modules to be imported and tested independently.

class mafw.devtools.toolchain.**HostTool**

Bases: *ToolChainTool*, ABC

Intermediate base for tools installed system-wide via pipx.

Concrete subclasses need only define *pipx_package_name* as a class-level attribute. This base class provides working default implementations of all abstract methods from *ToolChainTool*.

Subclasses may override individual methods for tool-specific behaviour (e.g. *UvTool* overrides *update()* and *verify()* to additionally synchronize pyproject.toml).

_ensure_pipx_available() → str

Locate the pipx executable on the system PATH.

Returns
Absolute path to the pipx executable.

Raises
DevtoolsError – If pipx is not found on the system PATH.

_get_pipx_version() → Version | None

Parse the installed version from `pipx list --json` output.

Queries pipx for its list of managed packages and extracts the version string for *pipx_package_name*.

Returns
The installed `Version`, or `None` if the package is not present in pipx.

Raises
DevtoolsError – If the pipx command fails or the JSON output has an unexpected structure.

bootstrap() → bool

Install the tool via `pipx install`.

Returns
`True` if installation was performed successfully.

Raises
DevtoolsError – If pipx is not available or installation fails.

detect_current_version() → Version | None

Detect the installed version from pipx metadata.

Runs `pipx list --json` and parses the JSON output to extract the version of this package.

Returns
The installed `Version`, or `None` if the package is not installed via pipx.

Raises
DevtoolsError – If pipx is not available or the JSON output cannot be parsed.

detect_latest_version() → Version

Query PyPI for the latest stable release.

Returns
The latest available `Version`.

Raises
DevtoolsError – If the PyPI query fails.

update() → bool

Upgrade the tool via `pipx upgrade`.

Compares the installed version before and after the upgrade to determine whether a change occurred.

Returns
`True` if the version changed, `False` if the tool was already at the latest version.

Raises
DevtoolsError – If pipx is not available or the upgrade fails.

verify() → list[*Issue*]

Verify configuration consistency.

Host tools that do not modify repository files have no cross-file consistency checks. Override in subclasses that need verification (e.g. uv checks its pyproject.toml specifier).

Returns
An empty list (no issues to report).

property category: Literal['host', 'project']

Tool category: `'host'` (pipx-managed).

property name: str

Human-readable tool identifier.

Defaults to *pipx_package_name*. Override in subclasses that need a different display name.

abstract property pipx_package_name: str

The package name as known to pipx (e.g. `'hatch'`, `'uv'`).

class mafw.devtools.toolchain.**Issue**(*tool_name: str*, *description: str*)

Bases: `object`

A configuration inconsistency detected during verification.

Each instance captures a single problem found by a tool's *ToolChainTool.verify()* method.

Parameters

- **tool_name** – Name of the tool that reported the issue.
- **description** – Human-readable explanation of the inconsistency.

class mafw.devtools.toolchain.**ProjectTool**(*project_root: Path | None = None*)

Bases: *ToolChainTool*, ABC

Intermediate base for tools managed via pyproject.toml.

Concrete subclasses need only define *package_name* and *section_path* as class-level attributes. The base class provides working default implementations of all abstract methods from *ToolChainTool*.

Subclasses override only what differs (e.g. custom update logic for ruff, `post_update` hooks for pytest/mypy/sphinx/pre-commit).

Parameters
project_root (*Path | None*) – Path to the project root directory containing `pyproject.toml`. Defaults to the current working directory.

static **_parse_resolution_failure**(*stderr: str*) → str | None

Extract the conflicting package name from resolver error output.

Parses hatch/uv resolver error messages to identify which package caused a dependency resolution failure. Returns the package name if found, or `None` if the error cannot be parsed.

Handles common patterns such as:

- `"Because only <package><=X.Y is available ..."`
- `"Because <package>>=X.Y depends on ..."`
- `"package <package> has no version that satisfies ..."`
- `"Could not find a version that satisfies the requirement <package>"`
- `"No matching distribution found for <package>"`

Package names may contain letters, digits, hyphens, underscores, and dots (e.g. `sphinxcontrib-external-links`, `ruamel.yaml`).

Parameters
stderr – The standard error output from a failed resolver command.

Returns
The conflicting package name, or `None` if not parseable.

Return type
str | None

_get_highest_python_version() → str

Return the highest supported Python version (cached after first call).

On the first invocation, reads `tool.mafw.supported-python` from pyproject.toml and caches the result. Subsequent calls return the cached value without re-reading the file, avoiding repeated TOML parsing across multiple tools.

Returns
The highest supported Python version as a dotted string (e.g. `"3.14"`).

Return type
str

Raises
DevtoolsError – If `tool.mafw.supported-python` is missing or empty in pyproject.toml.

_get_modifier() → *PyprojectModifier*

Create a `PyprojectModifier` for this project's pyproject.toml.

Returns
A fresh modifier instance (not yet loaded).

Return type
PyprojectModifier

_recreate_environment(*env_name: str*, *py_version: str | None = None*) → None

Validate dependency resolution via a temporary clone environment.

Instead of removing and recreating the real environment (which may be the active environment running `devtools`), this method creates a temporary clone environment in pyproject.toml, attempts to create it via `hatch env create`, and removes it afterwards. If the clone creation fails due to a dependency resolution conflict, the pyproject.toml is reverted (removing the clone declaration) and a descriptive error is raised.

The temporary environment inherits the `template` of the target environment, so it resolves the same dependency graph. Its name is `_toolchain_verify` to avoid collisions with real environments.

Parameters

- **env_name** – The hatch environment base name to validate (e.g. `'dev'`, `'types'`, `'hatch-test'`).
- **py_version** – Python version for matrix slot targeting (e.g. `'3.14'`). When provided, the clone is created for that specific Python version.

Raises
DevtoolsError – If the temporary environment creation fails (indicating a resolution conflict with the updated dependencies).

_revert() → None

Restore pyproject.toml to the content saved before *update()*.

If no content was saved (update was not called or already reverted), this method is a no-op.

bootstrap() → bool

Project tools cannot be bootstrapped independently.

They are installed automatically when syncing the appropriate Hatch environment.

Raises
DevtoolsError – Always, since project tools do not support standalone bootstrapping.

detect_current_version() → Version | None

Parse the >= lower bound from pyproject.toml.

Reads the dependency specifier for *package_name* from the section at *section_path* and extracts the version from its >= constraint using *LOWER_BOUND_RE*.

Returns
The currently configured lower-bound version, or `None` if the dependency cannot be found or lacks a >= specifier.

Return type
Version | None

detect_latest_version() → Version

Query PyPI for the latest stable release.

Delegates to *fetch_latest_version()* using *package_name*.

Returns
The latest non-pre-release version on PyPI.

Return type
Version

Raises
DevtoolsError – If the PyPI query fails.

update() → bool

Update the lower bound in pyproject.toml.

Compares the current lower bound with the latest PyPI version and updates the specifier if necessary. Saves the original file content before modification so that subclasses with `post_update()` hooks can revert on failure via *_revert()*.

Returns
`True` if the version was changed, `False` if already up to date.

Raises
DevtoolsError – If the update process fails.

verify() → list[*Issue*]

Return an empty list — default for single-location tools.

Most project tools are referenced in a single pyproject.toml section, so there is no cross-file consistency to check. Tools with multiple locations (e.g. ruff) override this method.

Returns
An empty list of issues.

Return type
list[*Issue*]

property category: Literal['host', 'project']

Tool category.

Returns
`"project"` — this tool is managed via pyproject.toml.

Return type
Category

abstract property env_name: str

Hatch environment base name where this tool lives.

This is the environment that gets recreated (via a temporary clone) during post-update validation. Examples: `'dev'`, `'types'`, `'hatch-test'`.

property name: str

Human-readable tool identifier.

Defaults to *package_name*. Override in subclasses that need a different display name.

abstract property package_name: str

PyPI package name (e.g. `'pytest'`, `'git-cliff'`).

abstract property section_path: str

Dot-separated TOML path to the dependency array.

Examples: `'project.optional-dependencies.dev'`, `'tool.hatch.envs.types.extra-dependencies'`.

class mafw.devtools.toolchain.**ToolChainTool**

Bases: ABC

Abstract interface that every managed development tool must implement.

Concrete subclasses represent individual tools (e.g. ruff, pytest, hatch) and provide the logic for bootstrapping, version detection, updating, and configuration verification.

abstractmethod bootstrap() → bool

Install the tool if not present.

Returns
`True` if installation was performed.

Raises
Exception – If the tool is already installed or installation fails.

abstractmethod detect_current_version() → Version | None

Return the currently installed or configured version.

Returns
The current `Version`, or `None` if the tool is not installed.

abstractmethod detect_latest_version() → Version

Return the latest available version from PyPI or the canonical source.

Returns
The latest `Version`.

Raises
Exception – If the version cannot be determined (e.g. network error).

post_update() → None

Hook executed after a successful update.

Override in subclasses to run validation (e.g. test suites, type checks) after the tool's version has been bumped. The default implementation is a no-op.

abstractmethod update() → bool

Update the tool to its latest version.

Returns
`True` if a configuration change was made, `False` if already up to date.

Raises
Exception – If the update process fails.

abstractmethod verify() → list[*Issue*]

Check configuration consistency and return any detected issues.

Returns
A list of *Issue* objects describing inconsistencies, or an empty list when the tool's configuration is consistent.

abstract property category: Literal['host', 'project']

Tool category: `'host'` (pipx-managed) or `'project'` (pyproject-managed).

abstract property name: str

Human-readable tool identifier (lowercase, e.g. `'ruff'`).

class mafw.devtools.toolchain.**ToolCheckResult**(*tool_name: str*, *category: str*, *current_version: Version | None*, *latest_version: Version | None*, *current_error: str | None = None*, *latest_error: str | None = None*)

Bases: `object`

Result of checking a single tool's version status.

Captures the current and latest versions along with any errors encountered during detection.

Parameters

- **tool_name** – Name of the tool that was checked.
- **category** – The tool's category (`"host"` or `"project"`).

- **current_version** – Currently installed/configured version, or `None` if detection failed.
- **latest_version** – Latest available version from the canonical source, or `None` if detection failed.
- **current_error** – Error message if current version detection failed.
- **latest_error** – Error message if latest version detection failed.

property in_sync: bool

Tool is in sync when both versions are known and equal.

Returns `False` if either version detection encountered an error or if the two versions differ.

class mafw.devtools.toolchain.**ToolRegistry**

Bases: `object`

Central registry that maintains all registered ToolChainTool instances.

Tools are stored in insertion order and indexed by name for O(1) lookup. Duplicate names are rejected to ensure each tool has a unique identifier within the registry.

Usage:

```
registry = ToolRegistry()
registry.register(my_tool)
all_tools = registry.all()
project_tools = registry.filter_by_category('project')
specific = registry.get('ruff')
```

all() → list[*ToolChainTool*]

Return all registered tools in insertion order.

Returns
A new list containing all tools in the order they were registered.

Return type
list[*toolchain.ToolChainTool*]

filter_by_category(*category: Literal['host', 'project']*) → list[*ToolChainTool*]

Return tools matching the given category, preserving insertion order.

Parameters
category – The category to filter by ("host" or "project").

Returns
A list of matching tools, or an empty list if none match.

Return type
list[*toolchain.ToolChainTool*]

get(*name: str*) → *ToolChainTool*

Retrieve a tool by its exact name (case-sensitive).

Parameters
name – The tool name to look up.

Returns
The matching *toolchain.ToolChainTool* instance.

Return type
toolchain.ToolChainTool

Raises
KeyError – If no tool with the given name is registered.

register(*tool:* ToolChainTool) → None

Add a tool to the registry.

The tool is appended to the internal list and indexed by name. If a tool with the same name already exists, a `ValueError` is raised and the registry state remains unchanged.

Parameters
tool – The tool instance to register.

Raises
ValueError – If a tool with the same name is already registered.

class mafw.devtools.toolchain.**ToolUpdateResult**(*tool_name: str*, *updated: bool*, *error: str | None = None*, *hook_error: str | None = None*, *revert_failed: bool = False*)

Bases: `object`

Result of processing a single tool during the update command.

Aggregates the outcome of calling *ToolChainTool.update()* and the optional *ToolChainTool.post_update()* hook.

Parameters

- **tool_name** – Name of the tool that was processed.
- **updated** – Whether the tool's configuration was changed.
- **error** – Error message if the update method raised an exception.
- **hook_error** – Error message if the post_update hook failed.
- **revert_failed** – Whether the rollback attempt after a hook failure also failed.

Modules

base	Abstract base class and data models for the toolchain management system.
default_registry	Default tool registry factory.
filtering	Tool filtering utility for the toolchain command group.
precommit_modifier	Pre-commit configuration modifier for toolchain management.
pypi	PyPI version query utility for toolchain management.
pyproject_modifier	Pyproject.toml modifier for toolchain dependency management.
registry	Central registry for ToolChainTool instances.
tools	Concrete tool implementations for the toolchain management system.

mafw.devtools.toolchain.base

Abstract base class and data models for the toolchain management system.

This module defines the *ToolChainTool* interface that every managed development tool must implement, the *ProjectTool* intermediate base class for tools managed via pyproject.toml, the *HostTool* intermediate base class for pipx-managed host tools, and supporting data models used by the CLI commands to aggregate and report results.

Module Attributes

LOWER_BOUND_RE	Regex to extract the version portion from a >= specifier.
Category	Tool category discriminator.

Classes

HostTool()	Intermediate base for tools installed system-wide via pipx.
Issue(tool_name, description)	A configuration inconsistency detected during verification.
ProjectTool([project_root])	Intermediate base for tools managed via pyproject.toml.
ToolChainTool()	Abstract interface that every managed development tool must implement.
ToolCheckResult(tool_name, category, ...[, ...])	Result of checking a single tool's version status.
ToolUpdateResult(tool_name, updated[, ...])	Result of processing a single tool during the update command.

class mafw.devtools.toolchain.base.**HostTool**

Bases: *ToolChainTool*, ABC

Intermediate base for tools installed system-wide via pipx.

Concrete subclasses need only define *pipx_package_name* as a class-level attribute. This base class provides working default implementations of all abstract methods from *ToolChainTool*.

Subclasses may override individual methods for tool-specific behaviour (e.g. *UvTool* overrides *update()* and *verify()* to additionally synchronize pyproject.toml).

_ensure_pipx_available() → str

Locate the pipx executable on the system PATH.

Returns
Absolute path to the pipx executable.

Raises
DevtoolsError – If pipx is not found on the system PATH.

_get_pipx_version() → Version | None

Parse the installed version from `pipx list --json` output.

Queries pipx for its list of managed packages and extracts the version string for *pipx_package_name*.

Returns
The installed `Version`, or `None` if the package is not present in pipx.

Raises
DevtoolsError – If the pipx command fails or the JSON output has an unexpected structure.

bootstrap() → bool

Install the tool via `pipx install`.

Returns
`True` if installation was performed successfully.

Raises
DevtoolsError – If pipx is not available or installation fails.

detect_current_version() → Version | None

Detect the installed version from pipx metadata.

Runs `pipx list --json` and parses the JSON output to extract the version of this package.

Returns
The installed `Version`, or `None` if the package is not installed via pipx.

Raises
DevtoolsError – If pipx is not available or the JSON output cannot be parsed.

detect_latest_version() → Version

Query PyPI for the latest stable release.

Returns
The latest available `Version`.

Raises
DevtoolsError – If the PyPI query fails.

update() → bool

Upgrade the tool via `pipx upgrade`.

Compares the installed version before and after the upgrade to determine whether a change occurred.

Returns
`True` if the version changed, `False` if the tool was already at the latest version.

Raises
DevtoolsError – If pipx is not available or the upgrade fails.

verify() → list[*Issue*]

Verify configuration consistency.

Host tools that do not modify repository files have no cross-file consistency checks. Override in subclasses that need verification (e.g. uv checks its pyproject.toml specifier).

Returns
An empty list (no issues to report).

property category: Literal['host', 'project']

Tool category: `'host'` (pipx-managed).

property name: str

Human-readable tool identifier.

Defaults to *pipx_package_name*. Override in subclasses that need a different display name.

abstract property pipx_package_name: str

The package name as known to pipx (e.g. `'hatch'`, `'uv'`).

class mafw.devtools.toolchain.base.**Issue**(*tool_name: str*, *description: str*)

Bases: `object`

A configuration inconsistency detected during verification.

Each instance captures a single problem found by a tool's *ToolChainTool.verify()* method.

Parameters

- **tool_name** – Name of the tool that reported the issue.
- **description** – Human-readable explanation of the inconsistency.

class mafw.devtools.toolchain.base.**ProjectTool**(*project_root: Path | None = None*)

Bases: *ToolChainTool*, ABC

Intermediate base for tools managed via pyproject.toml.

Concrete subclasses need only define *package_name* and *section_path* as class-level attributes. The base class provides working default implementations of all abstract methods from *ToolChainTool*.

Subclasses override only what differs (e.g. custom update logic for ruff, `post_update` hooks for pytest/mypy/sphinx/pre-commit).

Parameters
project_root (*Path | None*) – Path to the project root directory containing `pyproject.toml`. Defaults to the current working directory.

static _parse_resolution_failure(*stderr: str*) → str | None

Extract the conflicting package name from resolver error output.

Parses hatch/uv resolver error messages to identify which package caused a dependency resolution failure. Returns the package name if found, or `None` if the error cannot be parsed.

Handles common patterns such as:

- `"Because only <package><=X.Y is available ..."`
- `"Because <package>>=X.Y depends on ..."`
- `"package <package> has no version that satisfies ..."`
- `"Could not find a version that satisfies the requirement <package>"`
- `"No matching distribution found for <package>"`

Package names may contain letters, digits, hyphens, underscores, and dots (e.g. `sphinxcontrib-external-links`, `ruamel.yaml`).

Parameters
stderr – The standard error output from a failed resolver command.

Returns
The conflicting package name, or `None` if not parseable.

Return type
str | None

_get_highest_python_version() → str

Return the highest supported Python version (cached after first call).

On the first invocation, reads `tool.mafw.supported-python` from pyproject.toml and caches the result. Subsequent calls return the cached value without re-reading the file, avoiding repeated TOML parsing across multiple tools.

Returns
The highest supported Python version as a dotted string (e.g. `"3.14"`).

Return type
str

Raises
DevtoolsError – If `tool.mafw.supported-python` is missing or empty in pyproject.toml.

_get_modifier() → *PyprojectModifier*

Create a `PyprojectModifier` for this project's pyproject.toml.

Returns
A fresh modifier instance (not yet loaded).

Return type
PyprojectModifier

_recreate_environment(*env_name: str*, *py_version: str | None = None*) → None

Validate dependency resolution via a temporary clone environment.

Instead of removing and recreating the real environment (which may be the active environment running `devtools`), this method creates a temporary clone environment in pyproject.toml, attempts to create it via `hatch env create`, and removes it afterwards. If the clone creation fails due to a dependency resolution conflict, the pyproject.toml is reverted (removing the clone declaration) and a descriptive error is raised.

The temporary environment inherits the `template` of the target environment, so it resolves the same dependency graph. Its name is `_toolchain_verify` to avoid collisions with real environments.

Parameters

- **env_name** – The hatch environment base name to validate (e.g. `'dev'`, `'types'`, `'hatch-test'`).
- **py_version** – Python version for matrix slot targeting (e.g. `'3.14'`). When provided, the clone is created for that specific Python version.

Raises
DevtoolsError – If the temporary environment creation fails (indicating a resolution conflict with the updated dependencies).

_revert() → None

Restore pyproject.toml to the content saved before *update()*.

If no content was saved (update was not called or already reverted), this method is a no-op.

bootstrap() → bool

Project tools cannot be bootstrapped independently.

They are installed automatically when syncing the appropriate Hatch environment.

Raises
DevtoolsError – Always, since project tools do not support standalone bootstrapping.

detect_current_version() → Version | None

Parse the >= lower bound from pyproject.toml.

Reads the dependency specifier for *package_name* from the section at *section_path* and extracts the version from its >= constraint using *LOWER_BOUND_RE*.

Returns
The currently configured lower-bound version, or `None` if the dependency cannot be found or lacks a >= specifier.

Return type
Version | None

detect_latest_version() → Version

Query PyPI for the latest stable release.

Delegates to *fetch_latest_version()* using *package_name*.

Returns
The latest non-pre-release version on PyPI.

Return type
Version

Raises
DevtoolsError – If the PyPI query fails.

update() → bool

Update the lower bound in pyproject.toml.

Compares the current lower bound with the latest PyPI version and updates the specifier if necessary. Saves the original file content before modification so that subclasses with `post_update()` hooks can revert on failure via *_revert()*.

Returns
`True` if the version was changed, `False` if already up to date.

Raises
DevtoolsError – If the update process fails.

verify() → list[*Issue*]

Return an empty list — default for single-location tools.

Most project tools are referenced in a single pyproject.toml section, so there is no cross-file consistency to check. Tools with multiple locations (e.g. ruff) override this method.

Returns
An empty list of issues.

Return type
list[*Issue*]

property category: Literal['host', 'project']

Tool category.

Returns
"`project`" — this tool is managed via pyproject.toml.

Return type
Category

abstract property env_name: str

Hatch environment base name where this tool lives.

This is the environment that gets recreated (via a temporary clone) during post-update validation. Examples: `'dev'`, `'types'`, `'hatch-test'`.

property name: str

Human-readable tool identifier.

Defaults to *package_name*. Override in subclasses that need a different display name.

abstract property package_name: str

PyPI package name (e.g. `'pytest'`, `'git-cliff'`).

abstract property section_path: str

Dot-separated TOML path to the dependency array.

Examples: `'project.optional-dependencies.dev'`, `'tool.hatch.envs.types.extra-dependencies'`.

class mafw.devtools.toolchain.base.**ToolChainTool**

Bases: ABC

Abstract interface that every managed development tool must implement.

Concrete subclasses represent individual tools (e.g. ruff, pytest, hatch) and provide the logic for bootstrapping, version detection, updating, and configuration verification.

abstractmethod bootstrap() → bool

Install the tool if not present.

Returns
`True` if installation was performed.

Raises
Exception – If the tool is already installed or installation fails.

abstractmethod detect_current_version() → Version | None

Return the currently installed or configured version.

Returns
The current `Version`, or `None` if the tool is not installed.

abstractmethod detect_latest_version() → Version

Return the latest available version from PyPI or the canonical source.

Returns
The latest `Version`.

Raises
Exception – If the version cannot be determined (e.g. network error).

post_update() → None

Hook executed after a successful update.

Override in subclasses to run validation (e.g. test suites, type checks) after the tool's version has been bumped. The default implementation is a no-op.

abstractmethod update() → bool

Update the tool to its latest version.

Returns
`True` if a configuration change was made, `False` if already up to date.

Raises
Exception – If the update process fails.

abstractmethod verify() → list[*Issue*]

Check configuration consistency and return any detected issues.

Returns
A list of *Issue* objects describing inconsistencies, or an empty list when the tool's configuration is consistent.

abstract property category: Literal['host', 'project']

Tool category: `'host'` (pipx-managed) or `'project'` (pyproject-managed).

abstract property name: str

Human-readable tool identifier (lowercase, e.g. `'ruff'`).

class mafw.devtools.toolchain.base.**ToolCheckResult**(*tool_name: str*, *category: str*, *current_version: Version | None*, *latest_version: Version | None*, *current_error: str | None = None*, *latest_error: str | None = None*)

Bases: `object`

Result of checking a single tool's version status.

Captures the current and latest versions along with any errors encountered during detection.

Parameters

- **tool_name** – Name of the tool that was checked.
- **category** – The tool's category (`"host"` or `"project"`).
- **current_version** – Currently installed/configured version, or `None` if detection failed.
- **latest_version** – Latest available version from the canonical source, or `None` if detection failed.

- **current_error** – Error message if current version detection failed.
- **latest_error** – Error message if latest version detection failed.

property in_sync: bool

Tool is in sync when both versions are known and equal.

Returns `False` if either version detection encountered an error or if the two versions differ.

class mafw.devtools.toolchain.base.**ToolUpdateResult**(*tool_name: str*, *updated: bool*, *error: str | None = None*, *hook_error: str | None = None*, *revert_failed: bool = False*)

Bases: `object`

Result of processing a single tool during the update command.

Aggregates the outcome of calling *ToolChainTool.update()* and the optional *ToolChainTool.post_update()* hook.

Parameters

- **tool_name** – Name of the tool that was processed.
- **updated** – Whether the tool's configuration was changed.
- **error** – Error message if the update method raised an exception.
- **hook_error** – Error message if the post_update hook failed.
- **revert_failed** – Whether the rollback attempt after a hook failure also failed.

mafw.devtools.toolchain.base.**Category**

Tool category discriminator.

- "host": tools installed system-wide via pipx (e.g. hatch, uv).
- "project": tools whose version is managed through pyproject.toml.

alias of `Literal`['host', 'project']

mafw.devtools.toolchain.base.**LOWER_BOUND_RE: Final[Pattern[str]] = re.compile('>=\\s*([A-Za-z0-9.*!+]+)')**

Regex to extract the version portion from a >= specifier.

Shared across all project tools that parse pyproject.toml dependency specifiers. Defined here to avoid duplication in each tool module.

mafw.devtools.toolchain.default_registry

Default tool registry factory.

This module provides the *get_default_registry()* function, which creates a *ToolRegistry* pre-populated with all concrete *ToolChainTool* implementations in their canonical registration order.

CLI commands use this factory to obtain the registry of all managed tools without hardcoding individual tool references.

Functions

get_default_registry()	Create a `ToolRegistry` populated with all managed tools.

mafw.devtools.toolchain.default_registry.**get_default_registry**() → *ToolRegistry*

Create a `ToolRegistry` populated with all managed tools.

The tools are registered in the canonical order: host tools first (hatch, uv), followed by project tools (pytest, mypy, sphinx, ruff, pyupgrade, pre-commit, git-cliff, pip-audit).

Returns
A fully-populated registry ready for use by CLI commands.

Return type
toolchain.ToolRegistry

mafw.devtools.toolchain.filtering

Tool filtering utility for the toolchain command group.

This module provides *resolve_tools()*, a shared function used by all CLI subcommands to determine which tools to process based on the `--include-host`, `-i`/`--include`, and `-e`/`--exclude` options. It enforces mutual exclusivity between include and exclude filters, warns on unrecognized tool names, and preserves the original registration order from the registry.

Functions

resolve_tools(registry, *[, include, ...])	Filter the registry based on CLI options.

mafw.devtools.toolchain.filtering.**resolve_tools**(*registry:* ToolRegistry, *, *include: tuple[str, ...] = ()*, *exclude: tuple[str, ...] = ()*, *include_host: bool = False*) → list[*ToolChainTool*]

Filter the registry based on CLI options.

The filtering pipeline proceeds in order:

1. Reject mutually exclusive `include` and `exclude` arguments.
2. Start with all tools from the registry (preserving insertion order).
3. If *include_host* is `False`, remove tools with category `"host"`.
4. If *include* is non-empty, keep only tools whose name is in the set (emit a warning for each unrecognized name).
5. If *exclude* is non-empty, remove tools whose name is in the set (emit a warning for each unrecognized name).

Parameters

- **registry** – The central tool registry to query.
- **include** – Tool names to include exclusively. Only tools whose `name` matches one of these values will be returned.
- **exclude** – Tool names to exclude. Tools whose `name` matches one of these values will be removed from the result.
- **include_host** – When `False` (the default), tools with `category="host"` are excluded before any name-based filtering.

Returns
A list of tools in their original registration order.

Raises
DevtoolsError – If both *include* and *exclude* are non-empty (mutual exclusivity violation).

mafw.devtools.toolchain.precommit_modifier

Pre-commit configuration modifier for toolchain management.

This module provides the *PreCommitModifier* helper that reads, modifies, and writes `.pre-commit-config.yaml` files while preserving YAML comments, key ordering, and inline formatting. It is used by concrete *ToolChainTool* implementations (e.g. Ruff) to keep pre-commit hook revisions in sync with pyproject.toml.

The implementation uses a hybrid approach: `ruamel.yaml` in round-trip mode (`typ='rt'`) for parsing and validating the YAML structure, combined with string-level replacement for the actual `rev` field update. This guarantees that all bytes not related to the targeted `rev` value remain unchanged after a write operation.

Classes

PreCommitModifier()	Read, modify, and write `.pre-commit-config.yaml` preserving formatting.

class mafw.devtools.toolchain.precommit_modifier.**PreCommitModifier**

Bases: `object`

Read, modify, and write `.pre-commit-config.yaml` preserving formatting.

This class uses a hybrid approach: `ruamel.yaml` in round-trip mode for parsing and locating repository entries, and string-level replacement for the actual `rev` field update. This guarantees that all bytes not related to the updated `rev` value remain byte-for-byte identical — including sequence indentation, comments, and inline formatting that ruamel.yaml might otherwise normalize.

Typical usage:

```
modifier = PreCommitModifier()
modifier.read(Path('.pre-commit-config.yaml'))
modifier.update_rev(
    repo_url='https://github.com/astral-sh/ruff-pre-commit',
    new_rev='v0.15.12',
)
modifier.write(Path('.pre-commit-config.yaml'))
```

read(*path: Path*) → None

Load a `.pre-commit-config.yaml` file into memory.

Parameters

path (*Path*) – Path to the `.pre-commit-config.yaml` file.

Raises

DevtoolsError – If the file does not exist, cannot be read, or contains invalid YAML.

update_rev(*repo_url: str*, *new_rev: str*) → None

Update the `rev` field for a repository entry matching the given URL.

Locates the repository entry whose `repo` field matches *repo_url* and replaces its `rev` value with *new_rev*. The replacement is performed at the string level to guarantee byte-for-byte preservation of all other content.

Parameters

- **repo_url** (*str*) – The repository URL to search for in the `repos` list.
- **new_rev** (*str*) – The new revision string to set (e.g. `"v0.15.12"`). Must be non-empty and at most 128 characters.

Raises

DevtoolsError – If no data has been loaded, if the URL is not found, if the matched entry has no `rev` field, or if *new_rev* is empty or exceeds 128 characters.

write(*path: Path*) → None

Write the (possibly modified) configuration back to disk.

The output preserves all YAML comments, key ordering, and inline formatting from the original file. Only explicitly modified fields (via *update_rev()*) will differ from the original content.

Parameters
path (*Path*) – Path where the configuration should be written.

Raises
DevtoolsError – If no data has been loaded or if writing fails.

mafw.devtools.toolchain.precommit_modifier.**_extract_raw_rev**(*content: str*, *repo_url: str*) → str

Extract the literal rev value from raw YAML content for a given repo URL.

This avoids relying on ruamel.yaml's parsed representation, which may coerce values (e.g. `0000000` becomes integer `0` under YAML 1.1 octal rules). Instead, we locate the `rev:` line following the repo URL and extract the value as a raw string.

Parameters

- **content** – The raw YAML file content.
- **repo_url** – The repository URL whose rev to extract.

Returns
The literal rev value as it appears in the file.

Raises
DevtoolsError – If the rev line cannot be found.

mafw.devtools.toolchain.precommit_modifier.**_replace_rev_after_repo**(*content: str*, *repo_url: str*, *old_rev: str*, *new_rev: str*) → str

Replace the rev value in raw YAML content after a specific repo URL line.

This function finds the line containing the repo URL and then locates the first `rev:` line that follows it (before the next `- repo:` line or end of repos). It replaces only the old_rev value on that line.

Parameters

- **content** – The raw YAML file content.
- **repo_url** – The repository URL to locate.
- **old_rev** – The current rev value to replace.
- **new_rev** – The new rev value to set.

Returns
The modified content with only the rev value changed.

mafw.devtools.toolchain.precommit_modifier.**_MAX_REV_LENGTH: Final[int] = 128**

Maximum allowed length for a `rev` string value.

mafw.devtools.toolchain.pypi

PyPI version query utility for toolchain management.

This module provides a helper to query the PyPI JSON API for the latest release version of a given package. It is used by all concrete *ToolChainTool* implementations to satisfy the `detect_latest_version` contract.

Functions

fetch_latest_version(package_name, *[, ...])	Query the PyPI JSON API for the latest release version of a package.

mafw.devtools.toolchain.pypi.**fetch_latest_version**(*package_name: str*, *, *allow_prerelease: bool = False*) → Version

Query the PyPI JSON API for the latest release version of a package.

This function fetches the package metadata from PyPI, extracts the `info.version` field, and returns it as a `Version`. Pre-release versions are filtered out unless explicitly allowed.

When `allow_prerelease` is False and the latest version on PyPI is a pre-release, the function falls back to scanning `releases` keys to find the most recent stable release.

Parameters

- **package_name** (*str*) – The PyPI package name to query (e.g. `"ruff"`).
- **allow_prerelease** (*bool*) – If True, pre-release versions are acceptable.

Returns
The latest available version from PyPI.

Return type
Version

Raises
DevtoolsError – If the HTTP request fails, the response cannot be parsed, or no suitable version is found.

mafw.devtools.toolchain.pypi.**_PYPI_JSON_URL: Final[str] = 'https://pypi.org/pypi/{package}/json'**

URL template for the PyPI JSON API endpoint.

mafw.devtools.toolchain.pypi.**_REQUEST_TIMEOUT_S: Final[float] = 30.0**

HTTP request timeout in seconds for PyPI queries.

mafw.devtools.toolchain.pyproject_modifier

Pyproject.toml modifier for toolchain dependency management.

This module provides the *PyprojectModifier* helper that reads, modifies, and writes dependency specifiers in `pyproject.toml` while preserving all existing comments, formatting, and key ordering via `tomlkit`.

Multiple concrete *ToolChainTool* implementations delegate their `update` logic to this helper so that version-specifier manipulation is centralized and tested once.

Supported section paths:

- `project.dependencies`
- `project.optional-dependencies.<group>`
- `tool.hatch.envs.<env>.extra-dependencies`

Classes

PyprojectModifier(path)	Read, modify, and write dependency specifiers in `pyproject.toml`.

class mafw.devtools.toolchain.pyproject_modifier.**PyprojectModifier**(*path: Path*)

Bases: object

Read, modify, and write dependency specifiers in pyproject.toml.

The modifier preserves all TOML comments, formatting, and key ordering by using tomlkit for parsing and serialization.

Usage:

```
modifier = PyprojectModifier(Path('pyproject.toml'))
modifier.read()
specifier = modifier.get_dependency_specifier(
    'pytest', 'project.optional-dependencies.test'
)
modifier.update_lower_bound(
    'pytest', '9.1.0', 'project.optional-dependencies.test'
)
modifier.write()
```

Parameters
path (*Path*) – Path to the pyproject.toml file.

static _split_section_path(*section_path: str*) → list[str]

Split a section path into individual keys for document traversal.

Handles the special case of optional-dependencies and extra-dependencies where the key itself contains a hyphen and must not be split on it.

Known patterns: - project.dependencies → ["project", "dependencies"] - project.optional-dependencies.test → ["project", "optional-dependencies", "test"] - tool.hatch.envs.types.extra-dependencies → ["tool", "hatch", "envs", "types", "extra-dependencies"]

Parameters
section_path (*str*) – The dot-separated section path.

Returns
List of keys to traverse.

Return type
list[str]

_resolve_section(*section_path: str*) → Array

Navigate the TOML document to the dependency array at *section_path*.

Supported path formats:

- project.dependencies
- project.optional-dependencies.<group>
- tool.hatch.envs.<env>.extra-dependencies

Parameters
section_path (*str*) – Dot-separated path to the dependency array.

Returns
The tomlkit Array object containing the dependency strings.

Raises
DevtoolsError – If the section path does not exist in the document.

get_dependency_specifier(*dependency_name: str*, *section_path: str*) → str

Return the raw PEP 508 requirement string for a named dependency.

The dependency is located using PEP 503–normalized name comparison (case-insensitive, hyphens underscores dots).

Parameters

- **dependency_name** (*str*) – The package name to look up (e.g. `"ruff"`).
- **section_path** (*str*) – Dot-separated path to the dependency array (e.g. `"project.optional-dependencies.dev"`).

Returns
The raw requirement string as stored in the TOML array.

Return type
str

Raises
DevtoolsError – If the section does not exist or the dependency is not found within it.

read() → None

Load `pyproject.toml` from disk preserving formatting.

Raises
DevtoolsError – If the file does not exist or cannot be parsed.

update_lower_bound(*dependency_name: str*, *new_version: str*, *section_path: str*) → None

Replace the >= lower-bound version for a named dependency.

Only the version portion of the *first* >= specifier is replaced; all surrounding whitespace, additional specifiers, extras, and environment markers are preserved.

Parameters

- **dependency_name** (*str*) – The package name whose bound to update.
- **new_version** (*str*) – The new PEP 440 version string (e.g. `"9.1.0"`).
- **section_path** (*str*) – Dot-separated path to the dependency array.

Raises
DevtoolsError – If the section does not exist, the dependency is not found, or the dependency does not contain a >= specifier.

write() → None

Write the (possibly modified) document back to disk.

Preserves all comments, formatting, and key ordering for portions of the file that were not explicitly modified.

Raises
DevtoolsError – If no document has been loaded via *read()*.

mafw.devtools.toolchain.pyproject_modifier.**_extract_package_name**(*specifier: str*) → str

Extract the distribution name from a PEP 508 requirement string.

Handles extras (e.g. `"pandas[hdf5]>=2.2.3"`), environment markers (e.g. `'; python_version >= "3.14"'`), and bare names.

Parameters
specifier (*str*) – A PEP 508 requirement string.

Returns
The distribution name portion.

Return type
str

mafw.devtools.toolchain.pyproject_modifier.**_normalize_name**(*name: str*) → str

Normalize a Python package name per PEP 503.

Converts to lowercase and replaces runs of hyphens, underscores, and dots with a single hyphen, making name comparison canonical.

Parameters
name (*str*) – The raw package name (e.g. `"ruamel.yaml"`).

Returns
Normalized form (e.g. `"ruamel-yaml"`).

Return type
str

mafw.devtools.toolchain.pyproject_modifier.**_LOWER_BOUND_RE: Final[Pattern[str]] = re.compile('(>=)\\s*([A-Za-z0-9.*!+]+)')**

Regex matching a >= specifier and its version portion.

Group 1 captures the >= operator. Group 2 captures the version string to be replaced.

mafw.devtools.toolchain.pyproject_modifier.**_PEP503_NORMALIZE_RE: Final[Pattern[str]] = re.compile('[-_.]+')**

Regex used for PEP 503 package-name normalization.

mafw.devtools.toolchain.registry

Central registry for ToolChainTool instances.

The *ToolRegistry* maintains an ordered collection of all registered development tools, enabling discovery and iteration by the CLI commands. Tools are stored in insertion order and can be retrieved by name or filtered by category.

Classes

ToolRegistry()	Central registry that maintains all registered ToolChainTool instances.

class mafw.devtools.toolchain.registry.**ToolRegistry**

Bases: `object`

Central registry that maintains all registered ToolChainTool instances.

Tools are stored in insertion order and indexed by name for O(1) lookup. Duplicate names are rejected to ensure each tool has a unique identifier within the registry.

Usage:

```
registry = ToolRegistry()
registry.register(my_tool)
all_tools = registry.all()
project_tools = registry.filter_by_category('project')
specific = registry.get('ruff')
```

all() → list[*ToolChainTool*]

Return all registered tools in insertion order.

Returns
A new list containing all tools in the order they were registered.

Return type
list[*toolchain.ToolChainTool*]

filter_by_category(*category: Literal['host', 'project']*) → list[*ToolChainTool*]

Return tools matching the given category, preserving insertion order.

Parameters
category – The category to filter by ("host" or "project").

Returns
A list of matching tools, or an empty list if none match.

Return type
list[*toolchain.ToolChainTool*]

get(*name: str*) → *ToolChainTool*

Retrieve a tool by its exact name (case-sensitive).

Parameters
name – The tool name to look up.

Returns
The matching *toolchain.ToolChainTool* instance.

Return type
toolchain.ToolChainTool

Raises
KeyError – If no tool with the given name is registered.

register(*tool:* ToolChainTool) → None

Add a tool to the registry.

The tool is appended to the internal list and indexed by name. If a tool with the same name already exists, a `ValueError` is raised and the registry state remains unchanged.

Parameters
tool – The tool instance to register.

Raises
ValueError – If a tool with the same name is already registered.

mafw.devtools.toolchain.tools

Concrete tool implementations for the toolchain management system.

Each module in this package implements the *ToolChainTool* interface for a specific development tool. Tools are categorized as either "host" (managed via pipx) or "project" (managed via pyproject.toml).

class mafw.devtools.toolchain.tools.**GitCliffTool**(*project_root: Path | None = None*)

Bases: *ProjectTool*

Manage the git-cliff changelog generator version in pyproject.toml.

Git-cliff is declared in `project.optional-dependencies.dev` with a >= lower-bound specifier. This tool queries PyPI for the latest release and updates the lower bound accordingly.

All behaviour (version detection, update, verify, bootstrap) is provided by *ProjectTool*.

Parameters
project_root (*Path | None*) – Path to the project root directory containing `pyproject.toml`. Defaults to the current working directory.

property env_name: str

Hatch environment where git-cliff lives.

Returns
"dev"

Return type
str

property package_name: str

PyPI package name.

Returns
"git-cliff"

Return type
str

property section_path: str

TOML section path where git-cliff is declared.

Returns
"project.optional-dependencies.dev"

Return type
str

class mafw.devtools.toolchain.tools.**HatchTool**

Bases: *HostTool*

Manage hatch as a host tool installed via pipx.

This tool handles bootstrapping (installing hatch via pipx), detecting the currently installed version from pipx metadata, querying PyPI for the latest release, and upgrading via `pipx upgrade`.

Since hatch is a host-level tool that does not modify repository files, the `verify()` method always returns an empty list.

All method implementations are inherited from *HostTool*.

property pipx_package_name: str

The pipx package name for hatch.

class mafw.devtools.toolchain.tools.**MypyTool**(*project_root: Path | None = None*)

Bases: *ProjectTool*

Manage the mypy static type checker version in pyproject.toml.

Mypy is categorized as a *project* tool because its version is controlled through the `types` hatch environment extra-dependencies in `pyproject.toml`. After an update, the `types` environment is recreated and mypy type checking is executed at the highest supported Python version to validate compatibility. If the check fails, pyproject.toml is reverted to its pre-update state.

All standard behaviour (version detection, update, verify, bootstrap) is provided by *ProjectTool*. Only *post_update()* is overridden to run mypy type checking.

Parameters
project_root (*Path | None*) – Path to the project root directory containing `pyproject.toml`. Defaults to the current working directory.

post_update() → None

Recreate the types environment and run mypy type checking.

Performs the following steps after a successful version update:

1. Forces recreation of the `types` hatch environment so that the newly specified dependency versions are resolved and installed.
2. Determines the highest supported Python version dynamically from `tool.mafw.supported-python` in pyproject.toml.
3. Runs mypy via `hatch run types.py<version>:check`.

If environment recreation fails due to a dependency conflict, the conflicting package name is extracted and reported. If the type check fails, `pyproject.toml` is reverted to its pre-update state.

Raises
DevtoolsError – If environment recreation fails or the mypy check exits with a non-zero exit code.

property env_name: str

Hatch environment where mypy is validated.

Returns
"types"

Return type
str

property package_name: str

PyPI package name.

Returns
"mypy"

Return type
str

property section_path: str

TOML section path where mypy is declared.

Returns
"tool.hatch.envs.types.extra-dependencies"

Return type
str

class mafw.devtools.toolchain.tools.**PipAuditTool**(*project_root: Path | None = None*)

Bases: *ProjectTool*

Manage pip-audit's version lower bound in pyproject.toml.

pip-audit is categorized as a *project* tool because its version is controlled through the `dev` optional-dependencies group in `pyproject.toml`. There are no additional configuration files to keep in sync, so `verify()` always returns an empty list (inherited from *ProjectTool*).

All behaviour — bootstrapping, version detection, updating, and verification — is provided by the *ProjectTool* base class.

Parameters
project_root (*Path*) – Path to the project root directory containing `pyproject.toml`. Defaults to the current working directory.

property env_name: str

Hatch environment where pip-audit lives.

property package_name: str

PyPI package name for pip-audit.

property section_path: str

Dot-separated TOML path to the dependency array containing pip-audit.

class mafw.devtools.toolchain.tools.**PreCommitTool**(*project_root: Path | None = None*)

Bases: *ProjectTool*

Manage pre-commit's version lower bound in pyproject.toml.

pre-commit is categorized as a *project* tool because its version is controlled through the `dev` optional-dependencies group in `pyproject.toml`. After an update, the `dev` hatch environment is recreated to ensure fresh dependency resolution, and then `pre-commit run --all-files` is executed to validate compatibility. If the command fails, pyproject.toml is reverted to its pre-update state.

All standard behaviour (version detection, update, bootstrap) is provided by *ProjectTool*. The *post_update()* is overridden to run pre-commit hooks, and *verify()* explicitly returns an empty list since pre-commit is referenced in a single location.

Parameters
project_root (*Path | None*) – Path to the project root directory containing `pyproject.toml`. Defaults to the current working directory.

post_update() → None

Recreate the dev environment, then run pre-commit hooks; revert on failure.

Forces the recreation of the `dev` hatch environment to ensure the newly specified pre-commit version is resolved and installed. Then runs `pre-commit run --all-files` via the `dev` environment to validate compatibility. If the command fails (non-zero exit code), pyproject.toml is reverted to its saved pre-update content and a *DevtoolsError* is raised.

Raises
DevtoolsError – If environment recreation fails or if `pre-commit run --all-files` exits with a non-zero code.

verify() → list[*Issue*]

Return an empty list — pre-commit is referenced in a single location.

pre-commit has no cross-file consistency to verify (unlike ruff which appears in both pyproject.toml and `.pre-commit-config.yaml`).

Returns
An empty list of issues.

Return type
list[*toolchain.Issue*]

property env_name: str

Hatch environment where pre-commit hooks are run.

Returns
`"dev"`

Return type
str

property package_name: str

PyPI package name.

Returns
`"pre-commit"`

Return type
str

property section_path: str

TOML section path where pre-commit is declared.

Returns
`"project.optional-dependencies.dev"`

Return type
str

class mafw.devtools.toolchain.tools.**PytestTool**(*project_root: Path | None = None*)

Bases: *ProjectTool*

Manage pytest's version lower bound in pyproject.toml.

pytest is categorized as a *project* tool because its version is controlled through the `test` optional-dependencies group in `pyproject.toml`. After an update, the `test` hatch environment is recreated and

the full test suite is executed at the highest supported Python version to validate compatibility. If the tests fail, pyproject.toml is reverted to its pre-update state.

All standard behaviour (version detection, update, verify, bootstrap) is provided by *ProjectTool*. Only *post_update()* is overridden to run the test suite.

Parameters
project_root (*Path | None*) – Path to the project root directory containing `pyproject.toml`. Defaults to the current working directory.

post_update() → None

Recreate the test environment, then run the test suite; revert on failure.

Steps:

1. Force recreation of the `test` hatch environment so that the newly specified pytest version is resolved and installed fresh.
2. Determine the highest supported Python version dynamically from `tool.mafw.supported-python` in pyproject.toml.
3. Execute `hatch test -py <version> --without-integration --without-slow-integration` from the project root.
4. If the test suite fails (non-zero exit code), attempt to extract the conflicting package name via `_parse_resolution_failure`, revert pyproject.toml, and raise a descriptive error.

Raises
DevtoolsError – If the test suite fails after the update, or if environment recreation fails due to a resolution conflict.

property env_name: str

Hatch environment where pytest is validated.

Returns
"hatch-test"

Return type
str

property package_name: str

PyPI package name.

Returns
"pytest"

Return type
str

property section_path: str

TOML section path where pytest is declared.

Returns
"project.optional-dependencies.test"

Return type
str

class mafw.devtools.toolchain.tools.**RuffTool**(*project_root: Path | None = None*)

Bases: *ProjectTool*

Manage the ruff linter/formatter across pyproject.toml and pre-commit.

Ruff is categorized as a *project* tool. Its version lower bound lives in `project.optional-dependencies.dev` and the pre-commit hook revision is declared in `.pre-commit-config.yaml` under the `astral-sh/ruff-pre-commit` repository. Both must stay in sync.

The *update()* method writes both files (pyproject first, then pre-commit). On *post_update()* failure, both files are reverted to their pre-update contents: pyproject.toml via the inherited `_revert()` helper, and `.pre-commit-config.yaml` via the local `_saved_precommit` attribute.

Parameters
project_root (*Path | None*) – Path to the project root directory containing both `pyproject.toml` and `.pre-commit-config.yaml`. Defaults to the current working directory.

_get_precommit_version() → Version | None

Read the ruff version from the pre-commit config rev field.

Parses `.pre-commit-config.yaml`, locates the `astral-sh/ruff-pre-commit` repo entry, reads its `rev` field, strips the `v` prefix, and returns it as a `Version`.

Returns
The version from the pre-commit rev field, or `None` if the repo or rev cannot be found.

Return type
Version | None

_revert_all() → None

Restore both config files to their saved pre-update state.

Uses the parent class `_revert()` for pyproject.toml and manually restores `.pre-commit-config.yaml` from the saved content. Silently handles the case where saved content is not available.

_revert_precommit() → None

Restore .pre-commit-config.yaml to its saved pre-update state.

Silently handles the case where saved content is not available.

post_update() → None

Recreate the dev environment and run ruff-check and ruff-format.

Performs the following steps after a successful version update:

1. Forces recreation of the `dev` hatch environment so that the newly specified dependency versions are resolved and installed.
2. Determines the highest supported Python version dynamically from `tool.mafw.supported-python` in pyproject.toml.
3. Runs `hatch run dev.py<version>:ruff-check`.
4. Runs `hatch run dev.py<version>:ruff-format`.

If environment recreation fails due to a dependency conflict, both pyproject.toml and .pre-commit-config.yaml are reverted. If either ruff script fails, both files are also reverted.

Raises
DevtoolsError – If environment recreation fails or either ruff script exits with a non-zero exit code.

update() → bool

Update the ruff version in pyproject.toml and .pre-commit-config.yaml.

Performs both updates:

1. Sets the >= lower bound for ruff in `project.optional-dependencies.dev` to the latest PyPI version (delegated to the parent class).
2. Updates the `rev` field of the `astral-sh/ruff-pre-commit` repo entry in `.pre-commit-config.yaml` to `v{latest_version}`.

Before modifying, saves both file contents so they can be reverted by *post_update()* on failure. The parent class handles saving pyproject.toml via `_saved_content`; this method additionally saves the pre-commit config.

Returns
`True` if either file was changed, `False` if both are already up to date.

Return type
bool

Raises
DevtoolsError – If either file cannot be read or written.

verify() → list[*Issue*]

Compare ruff version in pyproject.toml with the pre-commit rev.

Reads the >= lower bound from `project.optional-dependencies.dev` and the `rev` field (stripped of `v` prefix) from the `astral-sh/ruff-pre-commit` repo entry. Returns an `Issue` if they do not match.

Returns
A list containing at most one *toolchain.Issue* if the two versions differ, or an empty list if they are consistent.

Return type
list[*toolchain.Issue*]

property env_name: str

Hatch environment where ruff is validated.

Returns
`"dev"`

Return type
str

property package_name: str

PyPI package name.

Returns
`"ruff"`

Return type
str

property section_path: str

TOML section path where ruff is declared.

Returns
`"project.optional-dependencies.dev"`

Return type
str

class mafw.devtools.toolchain.tools.**SphinxTool**(*project_root: Path | None = None*)

Bases: *ProjectTool*

Manage Sphinx's version lower bound in pyproject.toml.

Sphinx is categorized as a *project* tool because its version is controlled through the `doc` optional-dependencies group in `pyproject.toml`. After an update, the `doc` hatch environment is recreated and the documentation build is executed to validate compatibility. If the build fails, pyproject.toml is reverted to its pre-update state.

All standard behaviour (version detection, update, verify, bootstrap) is provided by *ProjectTool*. Only *post_update()* is overridden to run the documentation build.

Parameters
project_root (*Path | None*) – Path to the project root directory containing `pyproject.toml`. Defaults to the current working directory.

post_update() → None

Recreate the doc environment and run the documentation build.

Forces recreation of the `doc` hatch environment to ensure the newly specified dependency versions are resolved and installed. Then executes `hatch run dev.py3.14:doc` from the project root.

If the documentation build fails (non-zero exit code), the pyproject.toml is reverted to its saved pre-update content and a *DevtoolsError* is raised. The error message includes the conflicting package name when the failure is caused by a dependency resolution conflict.

Raises
DevtoolsError – If environment recreation or the documentation build fails after the update.

property env_name: str

Hatch environment where sphinx documentation is built.

Returns
`"dev"`

Return type
str

property package_name: str

PyPI package name for Sphinx.

Returns
`"sphinx"`

Return type
str

property section_path: str

Dot-separated TOML path to the dependency array containing Sphinx.

Returns
`"project.optional-dependencies.doc"`

Return type
str

class mafw.devtools.toolchain.tools.**UvTool**(*pyproject_path: Path = PosixPath('pyproject.toml')*)

Bases: *HostTool*

Toolchain management for the **uv** package manager.

uv is installed as a host tool via pipx and its version specifier is additionally tracked in pyproject.toml under the `hatch-uv` hatch environment so that hatch uses a compatible uv version.

All common host-tool operations (bootstrap, version detection, PyPI queries) are inherited from *HostTool*. This subclass overrides *update()* to additionally synchronize the pyproject.toml specifier and *verify()* to check specifier satisfaction.

Parameters
pyproject_path (*Path*) – Path to the `pyproject.toml` file. Defaults to `pyproject.toml` in the current working directory.

update() → bool

Upgrade uv via pipx and update the pyproject.toml specifier.

Performs two operations:

1. Delegates to *update()* which runs `pipx upgrade uv` and checks if the version changed.
2. If the version changed, updates the lower-bound version specifier for uv in the `[tool.hatch.envs.hatch-uv]` dependencies section of pyproject.toml.

Returns
`True` if the version changed, `False` if already up to date.

Raises
DevtoolsError – If pipx is not available, the upgrade fails, or the pyproject.toml modification fails.

verify() → list[*Issue*]

Check that the pipx-installed uv satisfies the declared specifier.

Compares the version installed via pipx against the version specifier declared in pyproject.toml under `[tool.hatch.envs.hatch-uv]`.

Returns
A list containing one *Issue* if the versions are out of sync, or an empty list if consistent.

property pipx_package_name: str

The pipx package name for uv.

Modules

git_cliff	Git-cliff toolchain tool implementation.
hatch	Hatch host tool implementation.
mypy	Concrete *ProjectTool* implementation for **mypy**.
pip_audit	PipAuditTool — concrete ProjectTool for pip-audit.
precommit	Concrete *ProjectTool* implementation for **pre-commit**.
pytest	Concrete *ProjectTool* implementation for **pytest**.
pyupgrade	PyUpgradeTool — concrete *ProjectTool* for pyupgrade.
ruff	RuffTool — concrete *ProjectTool* for ruff.
sphinx	Concrete *ProjectTool* implementation for **Sphinx**.
uv	Concrete *HostTool* implementation for **uv**.

mafw.devtools.toolchain.tools.git_cliff

Git-cliff toolchain tool implementation.

This module provides the *GitCliffTool* concrete implementation of *ProjectTool* for managing the `git-cliff` changelog generator dependency in `pyproject.toml`.

Git-cliff is a project tool whose version lower bound is declared in `project.optional-dependencies.dev`. It has no cross-file consistency concerns (unlike ruff which also appears in `.pre-commit-config.yaml`), so all behaviour is inherited from *ProjectTool* without overrides.

Classes

GitCliffTool([project_root])	Manage the git-cliff changelog generator version in pyproject.toml.

class mafw.devtools.toolchain.tools.git_cliff.**GitCliffTool**(*project_root: Path | None = None*)

Bases: *ProjectTool*

Manage the git-cliff changelog generator version in pyproject.toml.

Git-cliff is declared in `project.optional-dependencies.dev` with a >= lower-bound specifier. This tool queries PyPI for the latest release and updates the lower bound accordingly.

All behaviour (version detection, update, verify, bootstrap) is provided by *ProjectTool*.

Parameters
project_root (*Path | None*) – Path to the project root directory containing `pyproject.toml`. Defaults to the current working directory.

property env_name: str

Hatch environment where git-cliff lives.

Returns
`"dev"`

Return type
str

property package_name: str

PyPI package name.

Returns
`"git-cliff"`

Return type
str

property section_path: str

TOML section path where git-cliff is declared.

Returns
`"project.optional-dependencies.dev"`

Return type
str

mafw.devtools.toolchain.tools.hatch

Hatch host tool implementation.

This module provides the *HatchTool* concrete implementation of *HostTool* for managing hatch as a host tool installed via pipx.

Hatch is the build backend and project management tool used by MAFw. As a host tool, it lives outside the project's virtual environments and is managed through pipx commands. All operational logic (bootstrap, version detection, upgrade, verification) is inherited from *HostTool*.

Classes

HatchTool()	Manage hatch as a host tool installed via pipx.

class mafw.devtools.toolchain.tools.hatch.**HatchTool**

Bases: *HostTool*

Manage hatch as a host tool installed via pipx.

This tool handles bootstrapping (installing hatch via pipx), detecting the currently installed version from pipx metadata, querying PyPI for the latest release, and upgrading via `pipx upgrade`.

Since hatch is a host-level tool that does not modify repository files, the `verify()` method always returns an empty list.

All method implementations are inherited from *HostTool*.

property pipx_package_name: str

The pipx package name for hatch.

mafw.devtools.toolchain.tools.mypy

Concrete *ProjectTool* implementation for **mypy**.

Mypy is a project tool whose version lower bound is declared in `tool.hatch.envs.types.extra-dependencies`. This module manages the >= specifier for mypy and validates updates by running type checking via the `types` hatch environment.

Update logic (inherited from *ProjectTool*):

1. Query PyPI for the latest stable mypy release.
2. Compare against the current lower bound in pyproject.toml.
3. If a newer version exists, update the >= specifier.

Post-update validation (overridden here):

4. Recreate the `types` hatch environment to force fresh dependency resolution.
5. Determine the highest supported Python version dynamically from pyproject.toml.
6. Run mypy via `hatch run types.py<version>:check`.
7. If the type check fails, revert pyproject.toml to its previous content.

The revert pattern uses `_revert()` which restores the raw pyproject.toml content saved before the update.

Classes

MypyTool([project_root])	Manage the mypy static type checker version in pyproject.toml.

class mafw.devtools.toolchain.tools.mypy.**MypyTool**(*project_root: Path | None = None*)

Bases: *ProjectTool*

Manage the mypy static type checker version in pyproject.toml.

Mypy is categorized as a *project* tool because its version is controlled through the `types` hatch environment extra-dependencies in `pyproject.toml`. After an update, the `types` environment is recreated and mypy type checking is executed at the highest supported Python version to validate compatibility. If the check fails, pyproject.toml is reverted to its pre-update state.

All standard behaviour (version detection, update, verify, bootstrap) is provided by *ProjectTool*. Only *post_update()* is overridden to run mypy type checking.

Parameters
project_root (*Path | None*) – Path to the project root directory containing `pyproject.toml`. Defaults to the current working directory.

post_update() → None

Recreate the types environment and run mypy type checking.

Performs the following steps after a successful version update:

1. Forces recreation of the `types` hatch environment so that the newly specified dependency versions are resolved and installed.
2. Determines the highest supported Python version dynamically from `tool.mafw.supported-python` in pyproject.toml.
3. Runs mypy via `hatch run types.py<version>:check`.

If environment recreation fails due to a dependency conflict, the conflicting package name is extracted and reported. If the type check fails, `pyproject.toml` is reverted to its pre-update state.

Raises
DevtoolsError – If environment recreation fails or the mypy check exits with a non-zero exit code.

property env_name: str

Hatch environment where mypy is validated.

Returns
"types"

Return type
str

property package_name: str

PyPI package name.

Returns
"mypy"

Return type
str

property section_path: str

TOML section path where mypy is declared.

Returns
"tool.hatch.envs.types.extra-dependencies"

Return type
str

mafw.devtools.toolchain.tools.pip_audit

PipAuditTool — concrete ProjectTool for pip-audit.

This module implements the *PipAuditTool* class that manages the `pip-audit` dependency in the `dev` optional-dependencies of `pyproject.toml`. pip-audit is a vulnerability scanner for Python packages and is referenced in a single location, so no cross-file consistency verification is required.

Since pip-audit has no `post_update` hook or custom logic, its implementation is purely declarative — only the package name and section path differ from the *ProjectTool* defaults.

Classes

PipAuditTool([project_root])	Manage pip-audit's version lower bound in pyproject.toml.

class mafw.devtools.toolchain.tools.pip_audit.**PipAuditTool**(*project_root: Path | None = None*)

Bases: *ProjectTool*

Manage pip-audit's version lower bound in pyproject.toml.

pip-audit is categorized as a *project* tool because its version is controlled through the `dev` optional-dependencies group in `pyproject.toml`. There are no additional configuration files to keep in sync, so `verify()` always returns an empty list (inherited from *ProjectTool*).

All behaviour — bootstrapping, version detection, updating, and verification — is provided by the *ProjectTool* base class.

Parameters
project_root (*Path*) – Path to the project root directory containing `pyproject.toml`. Defaults to the current working directory.

property env_name: str

Hatch environment where pip-audit lives.

property package_name: str

PyPI package name for pip-audit.

property section_path: str

Dot-separated TOML path to the dependency array containing pip-audit.

mafw.devtools.toolchain.tools.precommit

Concrete *ProjectTool* implementation for **pre-commit**.

pre-commit is a project tool whose version lower bound is declared in `project.optional-dependencies.dev`. This module manages the >= specifier for pre-commit and validates updates by running `pre-commit run --all-files` within the recreated `dev` hatch environment.

Update logic (inherited from *ProjectTool*):

1. Query PyPI for the latest stable pre-commit release.
2. Compare against the current lower bound in pyproject.toml.
3. If a newer version exists, update the >= specifier.

Post-update validation (overridden here):

4. Force recreation of the `dev` hatch environment to ensure fresh dependency resolution with the new pre-commit version.
5. Run `pre-commit run --all-files` via the recreated dev environment.
6. If the command fails, revert pyproject.toml to its previous content.

The revert pattern uses `_revert()` which restores the raw pyproject.toml content saved before the update.

Classes

PreCommitTool([project_root])	Manage pre-commit's version lower bound in pyproject.toml.

class mafw.devtools.toolchain.tools.precommit.**PreCommitTool**(*project_root: Path | None = None*)

Bases: *ProjectTool*

Manage pre-commit's version lower bound in pyproject.toml.

pre-commit is categorized as a *project* tool because its version is controlled through the `dev` optional-dependencies group in `pyproject.toml`. After an update, the `dev` hatch environment is recreated to ensure fresh dependency resolution, and then `pre-commit run --all-files` is executed to validate compatibility. If the command fails, pyproject.toml is reverted to its pre-update state.

All standard behaviour (version detection, update, bootstrap) is provided by *ProjectTool*. The *post_update()* is overridden to run pre-commit hooks, and *verify()* explicitly returns an empty list since pre-commit is referenced in a single location.

Parameters

project_root (*Path | None*) – Path to the project root directory containing `pyproject.toml`. Defaults to the current working directory.

post_update() → None

Recreate the dev environment, then run pre-commit hooks; revert on failure.

Forces the recreation of the `dev` hatch environment to ensure the newly specified pre-commit version is resolved and installed. Then runs `pre-commit run --all-files` via the `dev` environment to

validate compatibility. If the command fails (non-zero exit code), pyproject.toml is reverted to its saved pre-update content and a *DevtoolsError* is raised.

Raises
DevtoolsError – If environment recreation fails or if `pre-commit run --all-files` exits with a non-zero code.

verify() → list[*Issue*]

Return an empty list — pre-commit is referenced in a single location.

pre-commit has no cross-file consistency to verify (unlike ruff which appears in both pyproject.toml and `.pre-commit-config.yaml`).

Returns
An empty list of issues.

Return type
list[*toolchain.Issue*]

property env_name: str

Hatch environment where pre-commit hooks are run.

Returns
`"dev"`

Return type
str

property package_name: str

PyPI package name.

Returns
`"pre-commit"`

Return type
str

property section_path: str

TOML section path where pre-commit is declared.

Returns
`"project.optional-dependencies.dev"`

Return type
str

mafw.devtools.toolchain.tools.pytest

Concrete *ProjectTool* implementation for **pytest**.

pytest is a project tool whose version lower bound is declared in `project.optional-dependencies.test`. This module manages the >= specifier for pytest and validates updates by running the full test suite via hatch.

Update logic (inherited from *ProjectTool*):

1. Query PyPI for the latest stable pytest release.
2. Compare against the current lower bound in pyproject.toml.
3. If a newer version exists, update the >= specifier.

Post-update validation (overridden here):

4. Recreate the `test` hatch environment to force fresh dependency resolution.
5. Determine the highest supported Python version from `tool.mafw.supported-python`.
6. Run the test suite via `hatch test -py <version> --without-integration --without-slow-integration`.

7. If the test suite fails, revert pyproject.toml to its previous content.

The revert pattern uses `_revert()` which restores the raw pyproject.toml content saved before the update.

Classes

PytestTool([project_root])	Manage pytest's version lower bound in pyproject.toml.

class mafw.devtools.toolchain.tools.pytest.**PytestTool**(*project_root: Path | None = None*)

Bases: *ProjectTool*

Manage pytest's version lower bound in pyproject.toml.

pytest is categorized as a *project* tool because its version is controlled through the `test` optional-dependencies group in `pyproject.toml`. After an update, the `test` hatch environment is recreated and the full test suite is executed at the highest supported Python version to validate compatibility. If the tests fail, pyproject.toml is reverted to its pre-update state.

All standard behaviour (version detection, update, verify, bootstrap) is provided by *ProjectTool*. Only *post_update()* is overridden to run the test suite.

Parameters
project_root (*Path | None*) – Path to the project root directory containing `pyproject.toml`. Defaults to the current working directory.

post_update() → None

Recreate the test environment, then run the test suite; revert on failure.

Steps:

1. Force recreation of the `test` hatch environment so that the newly specified pytest version is resolved and installed fresh.
2. Determine the highest supported Python version dynamically from `tool.mafw.supported-python` in pyproject.toml.
3. Execute `hatch test -py <version> --without-integration --without-slow-integration` from the project root.
4. If the test suite fails (non-zero exit code), attempt to extract the conflicting package name via `_parse_resolution_failure`, revert pyproject.toml, and raise a descriptive error.

Raises
DevtoolsError – If the test suite fails after the update, or if environment recreation fails due to a resolution conflict.

property env_name: str

Hatch environment where pytest is validated.

Returns
"hatch-test"

Return type
str

property package_name: str

PyPI package name.

Returns
"pytest"

Return type
str

property section_path: str

TOML section path where pytest is declared.

Returns
"project.optional-dependencies.test"

Return type
str

mafw.devtools.toolchain.tools.pyupgrade

PyUpgradeTool — concrete *ProjectTool* for pyupgrade.

This module implements the *PyUpgradeTool* class that manages the `pyupgrade` syntax modernizer across two configuration files:

- `pyproject.toml` — lower-bound >= specifier in `project.optional-dependencies.dev`
- `.pre-commit-config.yaml` — the `rev` field of the `https://github.com/asottile/pyupgrade` repository entry

Because pyupgrade is referenced in two places, the `verify()` method checks that the two declared versions are consistent. The `post_update()` hook recreates the `dev` hatch environment, then runs pyupgrade via `pre-commit run pyupgrade --all-files` at the highest supported Python version, and reverts **both** files if the command fails.

The tool follows the same cross-file synchronization pattern as *RuffTool*: the base class's `_saved_content` and `_revert()` handle pyproject.toml, while a separate `_saved_precommit` attribute tracks the pre-commit config file state.

Classes

PyUpgradeTool([project_root])	Manage pyupgrade across pyproject.toml and pre-commit.

class mafw.devtools.toolchain.tools.pyupgrade.**PyUpgradeTool**(*project_root: Path | None = None*)

Bases: *ProjectTool*

Manage pyupgrade across pyproject.toml and pre-commit.

pyupgrade is categorized as a *project* tool. Its version lower bound lives in `project.optional-dependencies.dev` and the pre-commit hook revision is declared in `.pre-commit-config.yaml` under the `asottile/pyupgrade` repository. Both must stay in sync.

The *update()* method writes both files (pyproject first, then pre-commit). On *post_update()* failure, both files are reverted to their pre-update contents: pyproject.toml via the inherited `_revert()` helper, and `.pre-commit-config.yaml` via the local `_saved_precommit` attribute.

Parameters
project_root (*Path | None*) – Path to the project root directory containing both `pyproject.toml` and `.pre-commit-config.yaml`. Defaults to the current working directory.

_get_precommit_version() → Version | None

Read the pyupgrade version from the pre-commit config rev field.

Parses `.pre-commit-config.yaml`, locates the `asottile/pyupgrade` repo entry, reads its `rev` field, strips the `v` prefix, and returns it as a `Version`.

Returns
The version from the pre-commit rev field, or `None` if the repo or rev cannot be found.

Return type
Version | None

`_revert_all()` → None

Restore both config files to their saved pre-update state.

Uses the parent class `_revert()` for pyproject.toml and manually restores `.pre-commit-config.yaml` from the saved content. Silently handles the case where saved content is not available.

`_revert_precommit()` → None

Restore .pre-commit-config.yaml to its saved pre-update state.

Silently handles the case where saved content is not available.

`post_update()` → None

Recreate the dev environment and run pyupgrade via pre-commit.

Performs the following steps after a successful version update:

1. Forces recreation of the `dev` hatch environment so that the newly specified dependency versions are resolved and installed.
2. Determines the highest supported Python version dynamically from `tool.mafw.supported-python` in pyproject.toml.
3. Runs `pre-commit run pyupgrade --all-files` via the dev environment to execute pyupgrade on all source files.

If environment recreation fails due to a dependency conflict, both pyproject.toml and .pre-commit-config.yaml are reverted. If pyupgrade fails, both files are also reverted.

Raises
DevtoolsError – If environment recreation fails or pyupgrade exits with a non-zero exit code.

`update()` → bool

Update the pyupgrade version in pyproject.toml and .pre-commit-config.yaml.

Performs both updates:

1. Sets the >= lower bound for pyupgrade in `project.optional-dependencies.dev` to the latest PyPI version (delegated to the parent class).
2. Updates the `rev` field of the `asottile/pyupgrade` repo entry in `.pre-commit-config.yaml` to `v{latest_version}`.

Before modifying, saves both file contents so they can be reverted by *post_update()* on failure. The parent class handles saving pyproject.toml via `_saved_content`; this method additionally saves the pre-commit config.

Returns
`True` if either file was changed, `False` if both are already up to date.

Return type
bool

Raises
DevtoolsError – If either file cannot be read or written.

`verify()` → list[*Issue*]

Compare pyupgrade version in pyproject.toml with the pre-commit rev.

Reads the >= lower bound from `project.optional-dependencies.dev` and the `rev` field (stripped of `v` prefix) from the `asottile/pyupgrade` repo entry. Returns an `Issue` if they do not match.

Returns
A list containing at most one *toolchain.Issue* if the two versions differ, or an empty list if they are consistent.

Return type
list[*toolchain.Issue*]

property env_name: str

Hatch environment where pyupgrade is validated.

Returns
"dev"

Return type
str

property package_name: str

PyPI package name.

Returns
"pyupgrade"

Return type
str

property section_path: str

TOML section path where pyupgrade is declared.

Returns
"project.optional-dependencies.dev"

Return type
str

mafw.devtools.toolchain.tools.pyupgrade._PRECOMMIT_REPO_URL: Final[str] = 'https://github.com/asottile/pyupgrade'

Repository URL for the pyupgrade pre-commit hook entry.

mafw.devtools.toolchain.tools.ruff

RuffTool — concrete *ProjectTool* for ruff.

This module implements the *RuffTool* class that manages the `ruff` linter/formatter across two configuration files:

- `pyproject.toml` — lower-bound >= specifier in `project.optional-dependencies.dev`
- `.pre-commit-config.yaml` — the `rev` field of the `https://github.com/astral-sh/ruff-pre-commit` repository entry

Because ruff is referenced in two places, the `verify()` method checks that the two declared versions are consistent. The `post_update()` hook recreates the `dev` hatch environment, then runs both the `ruff-check` and `ruff-format` hatch scripts at the highest supported Python version, and reverts **both** files if either script fails.

Unlike simpler project tools (e.g. git-cliff, pip-audit), RuffTool overrides *update()*, *verify()*, and *post_update()* because of the cross-file synchronization requirement. The base class's `_saved_content` and `_revert()` handle pyproject.toml, while a separate `_saved_precommit` attribute tracks the pre-commit config file state.

Classes

RuffTool([project_root])	Manage the ruff linter/formatter across pyproject.toml and pre-commit.

class mafw.devtools.toolchain.tools.ruff.**RuffTool**(*project_root: Path | None = None*)

Bases: *ProjectTool*

Manage the ruff linter/formatter across pyproject.toml and pre-commit.

Ruff is categorized as a *project* tool. Its version lower bound lives in `project.optional-dependencies.dev` and the pre-commit hook revision is declared in `.pre-commit-config.yaml` under the `astral-sh/ruff-pre-commit` repository. Both must stay in sync.

The *update()* method writes both files (pyproject first, then pre-commit). On *post_update()* failure, both files are reverted to their pre-update contents: pyproject.toml via the inherited `_revert()` helper, and `.pre-commit-config.yaml` via the local `_saved_precommit` attribute.

Parameters

project_root (*Path | None*) – Path to the project root directory containing both `pyproject.toml` and `.pre-commit-config.yaml`. Defaults to the current working directory.

_get_precommit_version() → Version | None

Read the ruff version from the pre-commit config rev field.

Parses `.pre-commit-config.yaml`, locates the `astral-sh/ruff-pre-commit` repo entry, reads its `rev` field, strips the `v` prefix, and returns it as a `Version`.

Returns

The version from the pre-commit rev field, or `None` if the repo or rev cannot be found.

Return type

Version | None

_revert_all() → None

Restore both config files to their saved pre-update state.

Uses the parent class `_revert()` for pyproject.toml and manually restores `.pre-commit-config.yaml` from the saved content. Silently handles the case where saved content is not available.

_revert_precommit() → None

Restore .pre-commit-config.yaml to its saved pre-update state.

Silently handles the case where saved content is not available.

post_update() → None

Recreate the dev environment and run ruff-check and ruff-format.

Performs the following steps after a successful version update:

1. Forces recreation of the `dev` hatch environment so that the newly specified dependency versions are resolved and installed.
2. Determines the highest supported Python version dynamically from `tool.mafw.supported-python` in pyproject.toml.
3. Runs `hatch run dev.py<version>:ruff-check`.
4. Runs `hatch run dev.py<version>:ruff-format`.

If environment recreation fails due to a dependency conflict, both pyproject.toml and .pre-commit-config.yaml are reverted. If either ruff script fails, both files are also reverted.

Raises

DevtoolsError – If environment recreation fails or either ruff script exits with a non-zero exit code.

update() → bool

Update the ruff version in pyproject.toml and .pre-commit-config.yaml.

Performs both updates:

1. Sets the >= lower bound for ruff in `project.optional-dependencies.dev` to the latest PyPI version (delegated to the parent class).
2. Updates the `rev` field of the `astral-sh/ruff-pre-commit` repo entry in `.pre-commit-config.yaml` to `v{latest_version}`.

Before modifying, saves both file contents so they can be reverted by *post_update()* on failure. The parent class handles saving pyproject.toml via `_saved_content`; this method additionally saves the pre-commit config.

Returns
`True` if either file was changed, `False` if both are already up to date.

Return type
bool

Raises
DevtoolsError – If either file cannot be read or written.

verify() → list[*Issue*]

Compare ruff version in pyproject.toml with the pre-commit rev.

Reads the >= lower bound from `project.optional-dependencies.dev` and the `rev` field (stripped of `v` prefix) from the `astral-sh/ruff-pre-commit` repo entry. Returns an `Issue` if they do not match.

Returns
A list containing at most one *toolchain.Issue* if the two versions differ, or an empty list if they are consistent.

Return type
list[*toolchain.Issue*]

property env_name: str

Hatch environment where ruff is validated.

Returns
`"dev"`

Return type
str

property package_name: str

PyPI package name.

Returns
`"ruff"`

Return type
str

property section_path: str

TOML section path where ruff is declared.

Returns
`"project.optional-dependencies.dev"`

Return type
str

mafw.devtools.toolchain.tools.ruff.**_PRECOMMIT_REPO_URL: Final[str] = 'https://github.com/astral-sh/ruff-pre-commit'**

Repository URL for the ruff pre-commit hook entry.

mafw.devtools.toolchain.tools.sphinx

Concrete *ProjectTool* implementation for **Sphinx**.

Sphinx is a project tool whose version lower bound is declared in `project.optional-dependencies.doc`. This module manages the >= specifier for Sphinx and validates updates by running the documentation build via hatch.

Update logic (inherited from *ProjectTool*):

1. Query PyPI for the latest stable Sphinx release.
2. Compare against the current lower bound in pyproject.toml.
3. If a newer version exists, update the >= specifier.

Post-update validation (overridden here):

4. Determine the highest supported Python version dynamically.
5. Recreate the `dev.py<version>` hatch environment to force fresh dependency resolution.
6. Run `hatch run dev.py<version>:doc` to build the documentation.
7. If the build fails, revert pyproject.toml to its previous content.

The revert pattern uses `_revert()` which restores the raw pyproject.toml content saved before the update.

Classes

SphinxTool([project_root])	Manage Sphinx's version lower bound in pyproject.toml.

class mafw.devtools.toolchain.tools.sphinx.**SphinxTool**(*project_root: Path | None = None*)

Bases: *ProjectTool*

Manage Sphinx's version lower bound in pyproject.toml.

Sphinx is categorized as a *project* tool because its version is controlled through the `doc` optional-dependencies group in `pyproject.toml`. After an update, the `doc` hatch environment is recreated and the documentation build is executed to validate compatibility. If the build fails, pyproject.toml is reverted to its pre-update state.

All standard behaviour (version detection, update, verify, bootstrap) is provided by *ProjectTool*. Only *post_update()* is overridden to run the documentation build.

Parameters
project_root (*Path | None*) – Path to the project root directory containing `pyproject.toml`. Defaults to the current working directory.

post_update() → None

Recreate the doc environment and run the documentation build.

Forces recreation of the `doc` hatch environment to ensure the newly specified dependency versions are resolved and installed. Then executes `hatch run dev.py3.14:doc` from the project root.

If the documentation build fails (non-zero exit code), the pyproject.toml is reverted to its saved pre-update content and a *DevtoolsError* is raised. The error message includes the conflicting package name when the failure is caused by a dependency resolution conflict.

Raises
DevtoolsError – If environment recreation or the documentation build fails after the update.

property env_name: str

Hatch environment where sphinx documentation is built.

Returns
`"dev"`

Return type
str

property package_name: str

PyPI package name for Sphinx.

Returns
"sphinx"

Return type
str

property section_path: str

Dot-separated TOML path to the dependency array containing Sphinx.

Returns
"project.optional-dependencies.doc"

Return type
str

mafw.devtools.toolchain.tools.uv

Concrete *HostTool* implementation for **uv**.

uv is a host tool installed system-wide via `pipx`. Its version must stay in sync with the `hatch-uv` dependency declared in pyproject.toml under `[tool.hatch.envs.hatch-uv]`.

Update logic:

1. Run `pipx upgrade uv` (via *update()*) to bring the pipx installation to the latest version.
2. Update the lower-bound specifier for uv in the hatch-uv environment dependencies section of `pyproject.toml`.

Verification logic:

Compare the pipx-installed version against the specifier declared in `pyproject.toml` and report an *Issue* if the installed version does not satisfy the declared constraint.

Classes

UvTool([pyproject_path])	Toolchain management for the **uv** package manager.

class mafw.devtools.toolchain.tools.uv.UvTool(*pyproject_path: Path = PosixPath('pyproject.toml')*)

Bases: *HostTool*

Toolchain management for the **uv** package manager.

uv is installed as a host tool via pipx and its version specifier is additionally tracked in pyproject.toml under the `hatch-uv` hatch environment so that hatch uses a compatible uv version.

All common host-tool operations (bootstrap, version detection, PyPI queries) are inherited from *HostTool*. This subclass overrides *update()* to additionally synchronize the pyproject.toml specifier and *verify()* to check specifier satisfaction.

Parameters
pyproject_path (*Path*) – Path to the `pyproject.toml` file. Defaults to `pyproject.toml` in the current working directory.

update() → bool

Upgrade uv via pipx and update the pyproject.toml specifier.

Performs two operations:

1. Delegates to *update()* which runs `pipx upgrade uv` and checks if the version changed.
2. If the version changed, updates the lower-bound version specifier for uv in the `[tool.hatch.envs.hatch-uv]` dependencies section of pyproject.toml.

Returns
`True` if the version changed, `False` if already up to date.

Raises
DevtoolsError – If pipx is not available, the upgrade fails, or the pyproject.toml modification fails.

verify() → list[*Issue*]

Check that the pipx-installed uv satisfies the declared specifier.

Compares the version installed via pipx against the version specifier declared in pyproject.toml under `[tool.hatch.envs.hatch-uv]`.

Returns
A list containing one *Issue* if the versions are out of sync, or an empty list if consistent.

property pipx_package_name: str

The pipx package name for uv.

mafw.devtools.toolchain.tools.uv.**_extract_specifier_from_requirement**(*requirement: str*) → str

Extract the version specifier portion from a PEP 508 requirement string.

Given a string like `"uv>=0.10"` or `"uv>0.10,<1.0"`, returns the specifier portion (e.g. `">=0.10"` or `">0.10,<1.0"`).

Parameters
requirement (*str*) – A raw PEP 508 requirement string.

Returns
The specifier portion of the requirement.

Return type
str

mafw.devtools.toolchain.tools.uv.**_HATCH_UV_SECTION: str = 'tool.hatch.envs.hatch-uv.dependencies'**

Section path in pyproject.toml where the uv dependency is declared.

mafw.devtools.toolchain.tools.uv.**_PYPROJECT_PATH: Path = PosixPath('pyproject.toml')**

Default path to the project's pyproject.toml file.

14.1.5 mafw.enumerators

Module provides a set of enumerators for dealing with standard tasks.

Classes

LogicalOp(*values)	Enumeration of supported logical operations.
LoopType(*values)	The loop strategy for the processor.
LoopingStatus(*values)	Enumerator to modify the looping cycle.
ProcessorExitStatus(*values)	The processor exit status enumerator class
ProcessorStatus(*values)	Enumerator to describe the status of a processor.

class mafw.enumerators.**LogicalOp**(**values*)

Bases: `Enum`

Enumeration of supported logical operations.

Added in version v1.3.0.

class mafw.enumerators.**LoopType**(**values*)

Bases: StrEnum

The loop strategy for the processor.

Each processor can be executed in one of the following modes:

1. **Single mode.** The process method is executed only once.
2. **For loop mode.** The process method is executed inside a for loop after the start and before the finish. The loop is based on a list of elements, the user **must** overload the *get_items()* method to define the list of items for the loop.
3. **Parallel for loop mode.** Conceptually identical to the for loop mode, but instead of a serial processing, items are shared among a pool of different threads. This looping mode is only available when running with a free-threading version of Python (recommended >= 3.14.4). In case free-threading is not available, the item processing will automatically fall back to the serial for loop mode.
4. **Parallel for loop with queue mode.** Similar to the parallel for loop, but processed items are pushed to a queue and handled by a single consumer thread. This allows serialized handling of *accept_item()* and *skip_item()* while still distributing the main processing over multiple workers. As with the parallel for loop, this execution mode is only available with free-threading and falls back to the serial for loop otherwise.
5. **While loop mode.** The process method is executed inside a while loop after the start and before the finish. The user **must** overload the *while_condition()* to define when to stop the loop.

static **_generate_next_value_**(*name*, *start*, *count*, *last_values*)

Return the lower-cased version of the member name.

ForLoop = 'for_loop'

Value for the for loop on item list execution.

ParallelForLoop = 'parallel_for_loop'

Value for the parallel for loop on item list execution.

ParallelForLoopWithQueue = 'parallel_for_loop_with_queue'

Value for the parallel for loop with queue execution.

SingleLoop = 'single'

Value for the single mode execution.

WhileLoop = 'while_loop'

Value for the while loop execution.

class mafw.enumerators.**LoopingStatus**(**values*)

Bases: IntEnum

Enumerator to modify the looping cycle.

In the case of a looping Processor, the user has the ability to slightly modify the looping structure using this enumerator.

In the *process()* the user can set the variable *looping_status* to one of the following values:

- *LoopingStatus.Continue*. It means that everything is working well and the loop cycle must go ahead as foreseen and the *accept_item()* will be invoked.
- *LoopingStatus.Skip*. The *skip_item()* will be called soon after the *process()* is finished. The status will be reset to Continue and the next item will be processed.
- *LoopingStatus.Abort*. The cycle is broken immediately.
- *LoopingStatus.Quit*. The cycle is broken immediately.

The last two options are apparently identical, but they offer the possibility to implement a different behaviour in the *finish()* method. When abort is used, then the *AbortProcessorException* will be raised. For example, the user can decide to rollback all changes if an abort as occurred or to save what done so far in case of a quit.

Abort = 3

Break the loop and force the outside container (*mafw.processor.ProcessorList*) to quit.

Continue = 1

The loop can continue

Quit = 4

Break the loop but let the outside container (*mafw.processor.ProcessorList*) to continue.

Skip = 2

Skip this item.

class mafw.enumerators.**ProcessorExitStatus**(**values*)

Bases: IntEnum

The processor exit status enumerator class

- Successful: means that the processor reached the end with success
- Failed: means that the processor did not reach the end with success
- Aborted: means that the user aborted the processor execution

Aborted = 3

The processor execution was aborted by the user

Failed = 2

The processor execution was failed

Successful = 1

The processor execution was successfully concluded

class mafw.enumerators.**ProcessorStatus**(**values*)

Bases: StrEnum

Enumerator to describe the status of a processor.

static _generate_next_value_(*name*, *start*, *count*, *last_values*)

Return the lower-cased version of the member name.

Finish = 'finishing'

Finished

Init = 'initializing'

Initialized

Run = 'processing'

Running

Start = 'starting'

Started

Unknown = 'unknown'

Unknown status

14.1.6 mafw.examples

A library of examples.

Modules

db_processors	The module provides some basic processors with DB interaction for demonstrating the basic functionalities.
importer_example	The module provides one concrete implementation of an Importer, as it was used in the autorad paper2.
loop_modifier	Examples on how the user can change the looping structure using the looping status.
multi_primary	Module demonstrates how to use multicolumn primary keys and foreign keys.
processor_list	The module provides some examples on how to use ProcessorList to combine several processors.
sum_processor	The module provides some examples for the user to develop their own processors.

mafw.examples.db_processors

The module provides some basic processors with DB interaction for demonstrating the basic functionalities.

Classes

CountStandardTables(*args, **kwargs)	A processor to count the number of standard tables
File(*args, **kwargs)	The Model class representing the table in the database
FillFileTableProcessor(*args, **kwargs)	Processor to fill a table with the content of a directory

Exceptions

FileDoesNotExist	Exception raised if the table corresponding to the File model does not exist.

exception mafw.examples.db_processors.**FileDoesNotExist**

Bases: Exception

Exception raised if the table corresponding to the File model does not exist.

class mafw.examples.db_processors.**CountStandardTables**(**args: Any*, ***kwargs: Any*)

Bases: *Processor*

A processor to count the number of standard tables

Processor parameters

- **n_tables**: The number of standard tables (default: -1)

Constructor parameters

Parameters

- **name** (*str, Optional*) – The name of the processor. If None is provided, the class name is used instead. Defaults to None.
- **description** (*str, Optional*) – A short description of the processor task. Defaults to the processor name.
- **config** (*dict, Optional*) – A configuration dictionary for this processor. Defaults to None.
- **looper** (LoopType, *Optional*) – Enumerator to define the looping type. Defaults to LoopType.ForLoop

- **user_interface** (*UserInterfaceBase, Optional*) – A user interface instance to be used by the processor to interact with the user.
- **timer** (*Timer, Optional*) – A timer object to measure process duration.
- **timer_params** (*dict, Optional*) – Parameters for the timer object.
- **database** (*Database, Optional*) – A database instance. Defaults to None.
- **database_conf** (*dict, Optional*) – Configuration for the database. Default to None.
- **remove_orphan_files** (*bool, Optional*) – Boolean flag to remove files on disc without a reference to the database. See *Standard tables* and *_remove_orphan_files()*. Defaults to True
- **replica_id** (*str, Optional*) – The replica identifier for the current processor.
- **create_standard_tables** (*bool, Optional*) – Boolean flag to create std tables on disk. Defaults to True When a nested steering configuration is loaded, this value can be overridden by the global create_standard_tables entry. Flat processor configurations keep the constructor value.
- **max_workers** (*int, Optional*) – Number of worker threads for parallel loops.
- **queue_size** (*int, Optional*) – Maximum size of the internal queue for the queue-based parallel loop.
- **queue_batch_size** (*int, Optional*) – Number of items processed per worker task in the queue-based parallel loop.
- **kwargs** – Keyword arguments that can be used to set processor parameters.

process()

Processes the current item.

This is the core of the Processor, where the user has to define the calculations required.

In parallel for loops, the method can optionally accept a *LoopItem* parameter if the user prefers not to rely on thread-local access to *processor.Processor.item*, *processor.Processor.i_item* and *processor.Processor.n_item*.

start() → None

Start method.

The user can overload this method, including all steps that should be performed at the beginning of the operation.

If the user decides to overload it, it should include a call to the super method.

class mafw.examples.db_processors.**File**(**args*, ***kwargs*)

Bases: *MAFwBaseModel*

The Model class representing the table in the database

DoesNotExist

alias of *FileDoesNotExist*

class mafw.examples.db_processors.**FillFileTableProcessor**(**args: Any*, ***kwargs: Any*)

Bases: *Processor*

Processor to fill a table with the content of a directory

Processor parameters

- **root_folder**: The root folder for the file listing (default: PosixPath('/tmp/mafw-docs-kpuvl5nc/v2.3.0_pdf'))

Constructor parameter:

Parameters
root_folder (*Path, Optional*) – ActiveParameter corresponding to the directory from where to start the recursive search

finish()

Transfers all the data to the File table via an atomic transaction.

format_progress_message()

Customizes the progress message with information about the current item.

The user can overload this method in order to modify the message being displayed during the process loop with information about the current item.

The user can access the current value, its position in the looping cycle and the total number of items using *processor.Processor.item*, *processor.Processor.i_item* and *processor.Processor.n_item*.

get_items() → list[Path]

Retrieves the list of files.

Insert or update the files from the root folder to the database

Returns
The list of full filename

Return type
list[Path]

process()

Add all information to the data list

start()

Starts the execution.

Be sure that the table corresponding to the File model exists. It it does already exists, it is not a problem.

mafw.examples.importer_example

The module provides one concrete implementation of an Importer, as it was used in the autorad paper2.

Classes

ImporterExample(*args, **kwargs)	An exemplary implementation of an importer processor.
InputElement(*args, **kwargs)	A model to store the input elements

Exceptions

InputElementDoesNotExist	Exception raised if the InputElement does not exist

exception mafw.examples.importer_example.**InputElementDoesNotExist**

Bases: DoesNotExist

Exception raised if the InputElement does not exist

class mafw.examples.importer_example.**ImporterExample**(**args: Any*, ***kwargs: Any*)

Bases: *Importer*

An exemplary implementation of an importer processor.

This importer subclass is looking for tif files in the input_folder and using the information stored in the filename, all required database fields will be obtained.

For this importer, we will use a filename parser including two compulsory elements and one optional ones. Those are:

- Sample name, in the form of sample_xyz, where xyz are three numerical digits (**compulsory**).
- Exposure time, in the form of expYh, where Y is the exposure time in hours. It can be an integer number or a floating number using . as decimal separator (**compulsory**).
- Resolution, in the form of rXYu, where XY is the readout granularity in micrometer. It is optional and its default is 25 µm if not provided.

This processor is subclassing:

1. The *start()*: in order to be sure that the database table is existing.
2. The *get_items()*: where the list of input files is retrieved. There we also verify that the table is still updated using the *verify_checksum()* and in case the user wants to process only new files, we will have to filter out from the list all items already stored in the database.
3. The *process()*: where we create a dictionary with the values to be stored in the database. This approach allows a much more efficient database transaction.
4. The *finish()*: where we actually do the database insert in a single go.
5. The *format_progress_message()*: to keep the user informed about the progress (optional).

Processor parameters

- **input_folder**: The input folder from where the images have to be imported. (default: '/tmp/mafw-docs-kpuvl5nc/v2.3.0_pdf')
- **parser_configuration**: The path to the TOML file with the filename parser configuration (default: 'parser_configuration.toml')
- **recursive**: Extend the search to sub-folder (default: True)

Constructor parameters

Parameters

- **name** (*str, Optional*) – The name of the processor. If None is provided, the class name is used instead. Defaults to None.
- **description** (*str, Optional*) – A short description of the processor task. Defaults to the processor name.
- **config** (*dict, Optional*) – A configuration dictionary for this processor. Defaults to None.
- **looper** (LoopType, *Optional*) – Enumerator to define the looping type. Defaults to LoopType.ForLoop
- **user_interface** (UserInterfaceBase, *Optional*) – A user interface instance to be used by the processor to interact with the user.
- **timer** (Timer, *Optional*) – A timer object to measure process duration.
- **timer_params** (*dict, Optional*) – Parameters for the timer object.
- **database** (*Database, Optional*) – A database instance. Defaults to None.
- **database_conf** (*dict, Optional*) – Configuration for the database. Default to None.
- **remove_orphan_files** (*bool, Optional*) – Boolean flag to remove files on disc without a reference to the database. See *Standard tables* and *_remove_orphan_files()*. Defaults to True
- **replica_id** (*str, Optional*) – The replica identifier for the current processor.

- **create_standard_tables** (*bool, Optional*) – Boolean flag to create std tables on disk. Defaults to True When a nested steering configuration is loaded, this value can be overridden by the global create_standard_tables entry. Flat processor configurations keep the constructor value.
- **max_workers** (*int, Optional*) – Number of worker threads for parallel loops.
- **queue_size** (*int, Optional*) – Maximum size of the internal queue for the queue-based parallel loop.
- **queue_batch_size** (*int, Optional*) – Number of items processed per worker task in the queue-based parallel loop.
- **kwargs** – Keyword arguments that can be used to set processor parameters.

finish() → None

The finish method overload.

Here is where we do the database insert with a on_conflict_replace to cope with the unique constraint.

format_progress_message() → None

Customizes the progress message with information about the current item.

The user can overload this method in order to modify the message being displayed during the process loop with information about the current item.

The user can access the current value, its position in the looping cycle and the total number of items using *processor.Processor.item*, *processor.Processor.i_item* and *processor.Processor.n_item*.

get_items() → Collection[Any]

Retrieves the list of element to be imported.

The base folder is provided in the configuration file, along with the recursive flags and all the filter options.

Returns
The list of items full file names to be processed.

Return type
list[Path]

process()

The process method overload.

This is where the whole list of files is scanned.

The current item is a filename, so we can feed it directly to the FilenameParser interpret command, to have it parsed. To maximise the efficiency of the database transaction, instead of inserting each file singularly, we are collecting them all in a list and then insert all of them in the *finish()* method.

In case the parsing is failing, then the element is skipped and an error message is printed.

start() → None

The start method.

The filename parser is ready to use because it has been already configured in the super method. We need to be sure that the input table exists, otherwise we create it from scratch.

class mafw.examples.importer_example.**InputElement**(**args*, ***kwargs*)

Bases: *MAFwBaseModel*

A model to store the input elements

DoesNotExist

alias of *InputElementDoesNotExist*

mafw.examples.loop_modifier

Examples on how the user can change the looping structure using the looping status.

Author
Bulgheroni Antonio (antonio.bulgheroni@ec.europa.eu)

Functions

is_prime(n)	Check if n is a prime number.

Classes

FindNPrimeNumber(*args, **kwargs)	An example of Processor to search for N prime numbers starting from a given starting integer.
FindPrimeNumberInRange(*args, **kwargs)	An example processor to find prime numbers in the defined interval from `start_from` to `stop_at`.
FindPrimeNumberInRangeParallel(*args, **kwargs)	An example processor to find prime numbers in parallel within the defined interval.
FindPrimeNumberInRangeParallelWithQueue(...)	An example processor to find prime numbers in parallel using a consumer queue.
FindPrimeNumberInRangeTunable(*args, **kwargs)	An example processor demonstrating loop tuning via *LoopTuningMixin*.
ModifyLoopProcessor(*args, **kwargs)	Example processor demonstrating how it is possible to change the looping structure.

class mafw.examples.loop_modifier.**FindNPrimeNumber**(**args: Any*, ***kwargs: Any*)

Bases: *Processor*

An example of Processor to search for N prime numbers starting from a given starting integer.

This processor is meant to demonstrate the use of a while_loop execution workflow.

Let us say we need to find 1000 prime numbers starting from 12347. One possible brute force approach to solve this problem is to start checking if the initial value is a prime number. If this is not the case, then check the next odd number. If it is the case, then add the current number to the list of found prime numbers and continue until the size of this list is 1000.

This is a perfect application for a while loop execution workflow.

Processor parameters

- **prime_num_to_find**: How many prime number we have to find (default: 100)
- **start_from**: From which number to start the search (default: 50)

Processor parameters:

Parameters

- **prime_num_to_find** (*int*) – The number of prime numbers to be found.
- **start_from** (*int*) – The initial integer number from where to start the search.

finish() → None

Overload of the finish method.

Remember: The finish method is called only once just after the last loop interaction. Always put a call to its *super* when you overload finish.

The loop is over, it means that the while condition was returning false, and now we can do something with our list of prime numbers.

format_progress_message() → None

Customizes the progress message with information about the current item.

The user can overload this method in order to modify the message being displayed during the process loop with information about the current item.

The user can access the current value, its position in the looping cycle and the total number of items using *processor.Processor.item*, *processor.Processor.i_item* and *processor.Processor.n_item*.

process() → None

The overload of the process method.

Remember: The process method is called inside the while loop. It has access to the looping parameters: *processor.Processor.i_item*, *processor.Processor.item* and *processor.Processor.n_item*.

In our specific case, the process contains another while loop. We start by checking if the current *processor.Processor.item* is a prime number or not. If so, then we have found the next prime number, we add it to the list, we increment by two units the value of *processor.Processor.item* and we leave the process method ready for the next iteration.

If *processor.Processor.item* is not prime, then increment it by 2 and check it again.

start() → None

The overload of the start method.

Remember: The start method is called just before the while loop is started. So all instructions in this method will be executed only once at the beginning of the process execution. Always put a call to its *super* when you overload start.

First, we empty the list of found prime numbers. It should not be necessary, but it makes the code more readable. Then set the *processor.Processor.n_item* to the total number of prime numbers we need to find. In this way, the progress bar will display useful progress.

If the start value is smaller than 2, then let's add 2 to the list of found prime number and set our first item to check at 3. In principle, we could already add 3 as well, but maybe the user wanted to find only 1 prime number, and we are returning a list with two, that is not what he was expecting.

Since prime numbers different from 2 can only be odd, if the starting number is even, increment it already by 1 unit.

while_condition() → bool

Define the while condition.

First, it checks if the prime_num_to_find is positive. Otherwise, it does not make sense to start. Then it will check if the length of the list with the already found prime numbers is enough. If so, then we can stop the loop return False, otherwise, it will return True and continue the loop.

Differently from the for_loop execution, we are responsible to assign the value to the looping variables *processor.Processor.i_item*, *processor.Processor.item* and *processor.Processor.n_item*.

In this case, we will use the *processor.Processor.i_item* to count how many prime numbers we have found and *processor.Processor.n_item* will be our target. In this way, the progress bar will work as expected.

In the while condition, we set the *processor.Processor.i_item* to the current length of the found prime number list.

Returns

True if the loop has to continue, False otherwise

prime_num_found: list[int]

The list with the found prime numbers

class mafw.examples.loop_modifier.**FindPrimeNumberInRange**(**args: Any*, ***kwargs: Any*)

Bases: *Processor*

An example processor to find prime numbers in the defined interval from `start_from` to `stop_at`.

This processor is meant to demonstrate the use of a for_loop execution workflow.

Let us say we want to select only the prime numbers in a user defined range. One possible brute force approach is to generate the list of integers between the range extremes and check if it is prime or not. If yes, then add it to the list of prime numbers, if not continue with the next element.

This is a perfect application for a loop execution workflow.

Processor parameters

- **start_from**: From which number to start the search (default: 50)
- **stop_at**: At which number to stop the search (default: 100)

Processor parameters:

Parameters

- **start_from** (*int*) – First element of the range under investigation.
- **stop_at** (*int*) – Last element of the range under investigation.

finish() → None

Overload of the finish method.

Remember: to call the super method when you overload the finish method.

In this case, we just print out some information about the prime number found in the range.

format_progress_message() → None

Customizes the progress message with information about the current item.

The user can overload this method in order to modify the message being displayed during the process loop with information about the current item.

The user can access the current value, its position in the looping cycle and the total number of items using *processor.Processor.item*, *processor.Processor.i_item* and *processor.Processor.n_item*.

get_items() → Collection[Any]

Overload of the get_items method.

This method must be overloaded when you select a for loop workflow.

Here we generate the list of odd numbers between the start and stop that we need to check. We also check that the stop is actually larger than the start, otherwise we print an error message, and we return an empty list of items.

Returns
A list of odd integer numbers between start_from and stop_at.

Return type
range[int]

process() → None

The process method.

In this case, it is very simple. We check if *processor.Processor.item* is a prime number, if so we added to the list, otherwise we let the loop continue.

start() → None

Overload of the start method.

Remember: to call the super method when you overload the start.

In this specific case, we just make sure that the list of found prime numbers is empty.

prime_num_found: list[int]

The list with the found prime numbers

class mafw.examples.loop_modifier.**FindPrimeNumberInRangeParallel**(**args: Any*, ***kwargs: Any*)

Bases: *Processor*

An example processor to find prime numbers in parallel within the defined interval.

This processor demonstrates the use of the parallel for loop execution workflow and highlights the need for thread-safe handling of shared state.

Processor parameters

- **start_from**: From which number to start the search (default: 50)
- **stop_at**: At which number to stop the search (default: 100)

Processor parameters:

Parameters

- **start_from** (*int*) – First element of the range under investigation.
- **stop_at** (*int*) – Last element of the range under investigation.

finish() → None

Print summary information about the prime numbers found.

format_progress_message() → None

Customizes the progress message with information about the current item.

The user can overload this method in order to modify the message being displayed during the process loop with information about the current item.

The user can access the current value, its position in the looping cycle and the total number of items using *processor.Processor.item*, *processor.Processor.i_item* and *processor.Processor.n_item*.

get_items() → Collection[Any]

Return a list of odd integer numbers between start_from and stop_at.

Returns
A list of odd integer numbers between start_from and stop_at.

Return type
list[int]

process() → None

Process a single item in parallel.

The update of shared results is guarded by a lock to remain thread-safe.

start() → None

Reset the list of found prime numbers.

_lock

Lock to protect shared list updates in parallel execution.

prime_num_found: list[int]

The list with the found prime numbers

class mafw.examples.loop_modifier.**FindPrimeNumberInRangeParallelWithQueue**(**args: Any*, ***kwargs: Any*)

Bases: *Processor*

An example processor to find prime numbers in parallel using a consumer queue.

The processing work is distributed across multiple threads, while accepting results happens on a single consumer thread, avoiding the need for explicit locks on shared state.

Processor parameters

- **start_from**: From which number to start the search (default: 50)
- **stop_at**: At which number to stop the search (default: 100)

Processor parameters:

Parameters

- **start_from** (*int*) – First element of the range under investigation.
- **stop_at** (*int*) – Last element of the range under investigation.

accept_item(*result:* LoopResult | *None* = *None*) → None

Store the prime number on the consumer thread.

finish() → None

Print summary information about the prime numbers found.

format_progress_message() → None

Customizes the progress message with information about the current item.

The user can overload this method in order to modify the message being displayed during the process loop with information about the current item.

The user can access the current value, its position in the looping cycle and the total number of items using *processor.Processor.item*, *processor.Processor.i_item* and *processor.Processor.n_item*.

get_items() → Collection[Any]

Return a range of odd integer numbers between start_from and stop_at.

Returns
A range of odd integer numbers between start_from and stop_at.

Return type
range[int]

process() → int | None

Process a single item in parallel.

The consumer thread will handle accepted results without additional locking.

start() → None

Reset the list of found prime numbers.

prime_num_found: list[int]

The list with the found prime numbers

class mafw.examples.loop_modifier.**FindPrimeNumberInRangeTunable**(**args: Any*, ***kwargs: Any*)

Bases: *LoopTuningMixin*, *Processor*

An example processor demonstrating loop tuning via *LoopTuningMixin*.

The processor can switch between serial and parallel for-loop workflows by setting the `loop_type` parameter in the steering file, together with the queue and worker settings.

Processor parameters

- **start_from**: From which number to start the search (default: 50)

- **stop_at**: At which number to stop the search (default: 100)
- **loop_type**: Loop type to use: for_loop, parallel_for_loop, parallel_for_loop_with_queue. (default: 'for_loop')
- **max_workers**: Number of worker threads for parallel loops. (default: 8)
- **queue_batch_size**: Batch size for queue processing in the queue-based parallel loop. (default: 1)
- **queue_size**: Queue size for the queue-based parallel loop. (default: 16)

Processor parameters:

Parameters

- **start_from** (*int*) – First element of the range under investigation.
- **stop_at** (*int*) – Last element of the range under investigation.

accept_item(*result:* LoopResult | *None* = *None*) → None

Store the prime number once the item has been accepted.

Parameters
result (mafw.models.LoopResult, *Optional*) – The loop result payload produced during processing.

finish() → None

Print summary information about the prime numbers found.

format_progress_message() → None

Customizes the progress message with information about the current item.

The user can overload this method in order to modify the message being displayed during the process loop with information about the current item.

The user can access the current value, its position in the looping cycle and the total number of items using *processor.Processor.item*, *processor.Processor.i_item* and *processor.Processor.n_item*.

get_items() → Collection[Any]

Return a range of odd integer numbers between start_from and stop_at.

Returns
A range of odd integer numbers between start_from and stop_at.

Return type
range[int]

process() → int | None

Process a single item and return the prime number payload if found.

start() → None

Reset the list of found prime numbers.

_lock

Lock to protect shared list updates in parallel execution.

prime_num_found: list[int]

The list with the found prime numbers.

class mafw.examples.loop_modifier.**ModifyLoopProcessor**(**args: Any*, ***kwargs: Any*)

Bases: *Processor*

Example processor demonstrating how it is possible to change the looping structure.

It is a looping processor where some events will be skipped, and at some point one event will trigger an abort.

Processor parameters

- **item_to_abort**: Item to abort (default: 65)
- **items_to_skip**: List of items to be skipped. (default: [12, 16, 25])
- **total_item**: Total item in the loop. (default: 100)

Processor Parameters:

Parameters

- **total_item** (*int*) – The total number of items
- **items_to_skip** (*list[int]*) – A list of items to skip.
- **item_to_abort** (*int*) – The item where to trigger an abort.

get_items() → range

Returns the list of items, the range from 0 to total_item.

process()

Processes the item

skip_item()

Add skipped item to the skipped item list.

start()

Resets the skipped item container.

skipped_items: [list[int]]

A list with the skipped items.

mafw.examples.loop_modifier.**is_prime**(*n: int*) → bool

Check if n is a prime number.

Parameters
n (*int*) – The integer number to be checked.

Returns
True if n is a prime number. False, otherwise.

Return type
bool

mafw.examples.multi_primary

Module demonstrates how to use multicolumn primary keys and foreign keys.

mafw.examples.processor_list

The module provides some examples on how to use ProcessorList to combine several processors.

Functions

run_processor_list_with_loop_modifier()	Example on deal with processors inside a processor list changing the loop structure.
run_simple_processor_list()	Simplest way to run several processors in a go.

mafw.examples.processor_list.**run_processor_list_with_loop_modifier**()

Example on deal with processors inside a processor list changing the loop structure.

In this example there are two processors, one that will run until the end and the other that will set the looping status to abort half way. The user can see what happens when the *ProcessorList* is executed.

mafw.examples.processor_list.**run_simple_processor_list**()

Simplest way to run several processors in a go.

mafw.examples.sum_processor

The module provides some examples for the user to develop their own processors.

Those implemented here are mainly used in the test suit.

Classes

AccumulatorProcessor(*args, **kwargs)	A processor to calculate the sum of the first n values via a looping approach.
GaussAdder(*args, **kwargs)	A processor to calculate the sum of the first n values via the so called *Gauss formula*.

class mafw.examples.sum_processor.**AccumulatorProcessor**(**args: Any*, ***kwargs: Any*)

Bases: *Processor*

A processor to calculate the sum of the first n values via a looping approach.

In mathematical terms, this processor solves this easy equation:

$$N = \sum_{i=0}^{n} i$$

by looping. It is a terribly inefficient approach, but it works as a demonstration of the looping structure.

The user can get the results by retrieving the *accumulated_value* parameter at the end of the processor execution.

Processor parameters

- **last_value**: Last value of the series (default: 100)

Constructor parameters:

Parameters

- **last_value** (*int*) – The *n* in the equation above. Defaults to 100
- **accumulated_value** (*int*) – The *N* in the equation above at the end of the process.

get_items() → list[int]

Returns the list of the first *last_value* integers.

process()

Increase the accumulated value by the current item.

start()

Resets the accumulated value to 0 before starting.

class mafw.examples.sum_processor.**GaussAdder**(**args: Any*, ***kwargs: Any*)

Bases: *Processor*

A processor to calculate the sum of the first n values via the so called *Gauss formula*.

In mathematical terms, this processor solves this easy equation:

$$N = \frac{n * (n-1)}{2}$$

without any looping

The user can get the results by retrieving the *sum_value* parameter at the end of the processor execution.

Processor parameters

- **last_value**: Last value of the series. (default: 100)

Constructor parameters:

Parameters

- **last_value** (*int*) – The n in the equation above. Defaults to 100
- **sum_value** (*int*) – The N in the equation above.

process()

Compute the sum using the Gauss formula.

start()

Sets the sum value to 0.

14.1.7 mafw.hookspecs

Defines the hook specification decorator bound the MAFw library.

Functions

register_db_model_modules()	Register database model modules
register_processors()	Register multiple processor classes
register_user_interfaces()	Register multiple user interfaces

mafw.hookspecs.**register_db_model_modules**() → list[str]

Register database model modules

mafw.hookspecs.**register_processors**() → list[*ProcessorClassProtocol*]

Register multiple processor classes

mafw.hookspecs.**register_user_interfaces**() → list[*UserInterfaceClassProtocol*]

Register multiple user interfaces

14.1.8 mafw.lazy_import

This module provides classes for lazy importing of plugins, ensuring thread-safe access and transparent usage.

Classes:

- LazyImportPlugin: A generic class for lazy importing of plugin classes.
- LazyImportProcessor: A specialised class for lazy importing of Processor plugins.
- LazyImportUserInterface: A specialised class for lazy importing of UserInterface plugins.

Protocols:

- ProcessorClassProtocol: Defines the expected interface for Processor classes.
- UserInterfaceClassProtocol: Defines the expected interface for UserInterface classes.

Added in version v2.0.0.

Module Attributes

T	The class type to be used for the generic lazy import plugin.
R	The instance type to be used for the generic lazy import plugin.

Classes

LazyImportPlugin(module, class_name)	Proxy object that lazily imports a plugin class only when accessed.
LazyImportProcessor(module, class_name)	Lazy import proxy for Processor classes.
LazyImportUserInterface(module, class_name, ...)	Lazy import proxy for UserInterface classes.
ProcessorClassProtocol(*args, **kwargs)	Protocol for Processor classes.
UserInterfaceClassProtocol(*args, **kwargs)	Protocol for UserInterface classes.

class mafw.lazy_import.**LazyImportPlugin**(*module: str*, *class_name: str*)

Bases: Generic[*T*, *R*], ABC

Proxy object that lazily imports a plugin class only when accessed. Thread-safe and transparent to the user.

Added in version v2.0.0.

Constructor parameter:

Parameters

- **module** (*str*) – The module name where the class is located.
- **class_name** (*str*) – The name of the class to be lazily imported.

_load() → type[*T*]

Load the class from the specified module.

Returns
The loaded class type.

Return type
type[*T*]

abstractmethod _post_load(*cls: type[T]*) → type[*T*]

Perform operations after loading the class.

Parameters
cls (*type[T]*) – The class type that has been loaded.

Returns
The class type after post-load operations.

Return type
type[*T*]

class mafw.lazy_import.**LazyImportProcessor**(*module: str*, *class_name: str*)

Bases: *LazyImportPlugin*[*Processor*, *Processor*]

Lazy import proxy for Processor classes.

Added in version v2.0.0.

Constructor parameter:

Parameters

- **module** (*str*) – The module name where the class is located.
- **class_name** (*str*) – The name of the class to be lazily imported.

_post_load(*cls: type[*Processor*]*) → type[*Processor*]

Perform operations after loading the Processor class.

Parameters
cls (*type[processor.Processor]*) – The Processor class type that has been loaded.

Returns
The Processor class type after post-load operations.

Return type
type[*processor.Processor*]

class mafw.lazy_import.**LazyImportUserInterface**(*module: str*, *class_name: str*, *ui_name: str*)

Bases: *LazyImportPlugin*[*UserInterfaceBase*, *UserInterfaceBase*]

Lazy import proxy for UserInterface classes.

Added in version v2.0.0.

Constructor parameter:

Parameters

- **module** (*str*) – The module name where the class is located.
- **class_name** (*str*) – The name of the class to be lazily imported.
- **ui_name** (*str*) – The expected name of the user interface.

_post_load(*cls: type[*UserInterfaceBase*]*) → type[*UserInterfaceBase*]

Perform operations after loading the UserInterface class.

Parameters
cls (*type[UserInterfaceBase]*) – The UserInterface class type that has been loaded.

Returns
The UserInterface class type after post-load operations.

Return type
type[*UserInterfaceBase*]

Raises
ValueError – If the class name is inconsistent with the expected name.

class mafw.lazy_import.**ProcessorClassProtocol**(**args*, ***kwargs*)

Bases: Protocol

Protocol for Processor classes.

Added in version v2.0.0.

Variables

- **plugin_name** – The name of the plugin.
- **plugin_qualname** – The qualified name of the plugin.
- **__name__** – The name of the class.

filter_schema() → *FilterSchema* | None

Return the filter schema.

parameter_schema() → list[*ParameterSchema*]

Return the parameter schema.

processor_schema() → *ProcessorSchema*

Return the processor schema.

class mafw.lazy_import.**R**

The instance type to be used for the generic lazy import plugin.

alias of TypeVar('R')

class mafw.lazy_import.**T**

The class type to be used for the generic lazy import plugin.

alias of TypeVar('T')

class mafw.lazy_import.**UserInterfaceClassProtocol**(**args*, ***kwargs*)

Bases: Protocol

Protocol for UserInterface classes.

Added in version v2.0.0.

Variables

- **plugin_name** – The name of the plugin.
- **plugin_qualname** – The qualified name of the plugin.
- **name** – The name of the user interface.

14.1.9 mafw.mafw_errors

Module defines MAFw exceptions

Exceptions

AbortProcessorException	Exception raised during the execution of a processor requiring immediate exit.
CyclicGroupIssue(node)	Issue raised when cyclic dependencies exist within processor groups.
DuplicateReplicaIssue(base, replica)	Issue raised when a processor declares duplicate replica identifiers.
InvalidConfigurationError	Error with the configuration of a processor
InvalidFilterConditionIssue(processor, ...)	Issue raised when a filter condition is incomplete or invalid.
InvalidFilterLogicIssue(processor, target, *)	Issue raised when a filter logic expression is invalid.
InvalidSteeringFile	Exception raised when validating an invalid steering file
MAFwException	Base class for MAFwException
MissingAttribute	Exception raised when an attempt is made to execute a statement without a required parameter/attributes
MissingDatabase	Exception raised when a processor requiring a database connection is being operated without a database
MissingOptionalDependency	UserWarning raised when an optional dependency is required
MissingOverloadedMethod	Warning issued when the user did not overload a required method.
MissingProcessorsToRunIssue()	Issue raised when the steering file does not declare `processors_to_run`.
MissingSQLStatement	Exception raised when a Trigger is created without any SQL statements.
MissingSuperCall	Warning issued when the user did not invoke the super method for some specific processor methods.
MissingUserInterfaceSectionIssue()	Issue raised when the `UserInterface` section is missing.
ModelError	Exception raised when an error in a DB Model class occurs
ParserConfigurationError	Exception raised when an error occurred during the configuration of a filename parser

continues on next page

Table 14.107 – continued from previous page

ParsingError	Exception raised when a regular expression parsing failed
PlotterMixinNotInitialized	Exception raised when a plotter mixin has not properly initialized
ProcessorParameterError	Error with a processor parameter
ProcessorsToRunNotListIssue()	Issue raised when `processors_to_run` is not a list.
RunnerNotInitialized	Exception raised when attempting to run a not initialized Runner.
SteeringFileDeprecation	Warning issued when a legacy deprecated configuration is used in the steering file.
UnknownDBEngine	Exception raised when the user provided an unknown db engine
UnknownGroupMemberIssue(group, member)	Issue raised when a group references an unknown processor or nested group.
UnknownProcessor	Exception raised when an attempt is made to create an unknown processor
UnknownProcessorGroup	Exception raised when an attempt is made to create an unknown processor group
UnknownProcessorsToRunEntryIssue(entry)	Issue raised when a processor reference in `processors_to_run` cannot be resolved.
UnsupportedDatabaseError	Error raised when a feature is not supported by the database.
ValidationIssue(message)	Base class for steering validation issues.

exception mafw.mafw_errors.**AbortProcessorException**

Bases: *MAFwException*

Exception raised during the execution of a processor requiring immediate exit.

exception mafw.mafw_errors.**CyclicGroupIssue**(*node: str*)

Bases: *ValidationIssue*

Issue raised when cyclic dependencies exist within processor groups.

exception mafw.mafw_errors.**DuplicateReplicaIssue**(*base: str*, *replica: str*)

Bases: *ValidationIssue*

Issue raised when a processor declares duplicate replica identifiers.

exception mafw.mafw_errors.**InvalidConfigurationError**

Bases: *MAFwException*

Error with the configuration of a processor

exception mafw.mafw_errors.**InvalidFilterConditionIssue**(*processor: str*, *model: str*, *condition: str*, ***, *field_name: str | None = None*, *reason: str | None = None*)

Bases: *ValidationIssue*

Issue raised when a filter condition is incomplete or invalid.

exception mafw.mafw_errors.**InvalidFilterLogicIssue**(*processor: str*, *target: str*, ***, *detail: str | None = None*)

Bases: *ValidationIssue*

Issue raised when a filter logic expression is invalid.

exception mafw.mafw_errors.**InvalidSteeringFile**

Bases: *MAFwException*

Exception raised when validating an invalid steering file

exception mafw.mafw_errors.**MAFwException**

Bases: Exception

Base class for MAFwException

exception mafw.mafw_errors.**MissingAttribute**

Bases: *MAFwException*

Exception raised when an attempt is made to execute a statement without a required parameter/attributes

exception mafw.mafw_errors.**MissingDatabase**

Bases: *MAFwException*

Exception raised when a processor requiring a database connection is being operated without a database

exception mafw.mafw_errors.**MissingOptionalDependency**

Bases: UserWarning

UserWarning raised when an optional dependency is required

exception mafw.mafw_errors.**MissingOverloadedMethod**

Bases: UserWarning

Warning issued when the user did not overload a required method.

It is a warning and not an error because the execution framework might still work, but the results might be different from what is expected.

exception mafw.mafw_errors.**MissingProcessorsToRunIssue**

Bases: *ValidationIssue*

Issue raised when the steering file does not declare processors_to_run.

exception mafw.mafw_errors.**MissingSQLStatement**

Bases: *MAFwException*

Exception raised when a Trigger is created without any SQL statements.

exception mafw.mafw_errors.**MissingSuperCall**

Bases: UserWarning

Warning issued when the user did not invoke the super method for some specific processor methods.

Those methods (like *start()* and *finish()*) have not empty implementation also in the base class, meaning that if the user forgets to call *super* in their overloads, then the basic implementation will be gone.

It is a warning and not an error because the execution framework might sill work, but the results might be different from what is expected.

exception mafw.mafw_errors.**MissingUserInterfaceSectionIssue**

Bases: *ValidationIssue*

Issue raised when the UserInterface section is missing.

exception mafw.mafw_errors.**ModelError**

Bases: *MAFwException*

Exception raised when an error in a DB Model class occurs

exception mafw.mafw_errors.**ParserConfigurationError**

Bases: *MAFwException*

Exception raised when an error occurred during the configuration of a filename parser

exception mafw.mafw_errors.**ParsingError**

Bases: *MAFwException*

Exception raised when a regular expression parsing failed

exception mafw.mafw_errors.**PlotterMixinNotInitialized**

Bases: *MAFwException*

Exception raised when a plotter mixin has not properly initialized

exception mafw.mafw_errors.**ProcessorParameterError**

Bases: *MAFwException*

Error with a processor parameter

exception mafw.mafw_errors.**ProcessorsToRunNotListIssue**

Bases: *ValidationIssue*

Issue raised when `processors_to_run` is not a list.

exception mafw.mafw_errors.**RunnerNotInitialized**

Bases: *MAFwException*

Exception raised when attempting to run a not initialized Runner.

exception mafw.mafw_errors.**SteeringFileDeprecation**

Bases: `UserWarning`

Warning issued when a legacy deprecated configuration is used in the steering file.

It is a warning and not an error because the execution framework might still work, but the configuration may be removed in future versions.

exception mafw.mafw_errors.**UnknownDBEngine**

Bases: *MAFwException*

Exception raised when the user provided an unknown db engine

exception mafw.mafw_errors.**UnknownGroupMemberIssue**(*group: str*, *member: str*)

Bases: *ValidationIssue*

Issue raised when a group references an unknown processor or nested group.

exception mafw.mafw_errors.**UnknownProcessor**

Bases: *MAFwException*

Exception raised when an attempt is made to create an unknown processor

exception mafw.mafw_errors.**UnknownProcessorGroup**

Bases: *MAFwException*

Exception raised when an attempt is made to create an unknown processor group

exception mafw.mafw_errors.**UnknownProcessorsToRunEntryIssue**(*entry: str*)

Bases: *ValidationIssue*

Issue raised when a processor reference in `processors_to_run` cannot be resolved.

exception mafw.mafw_errors.**UnsupportedDatabaseError**

Bases: `Exception`

Error raised when a feature is not supported by the database.

exception mafw.mafw_errors.**ValidationIssue**(*message: str*)

Bases: *InvalidSteeringFile*

Base class for steering validation issues.

14.1.10 mafw.models

Model helpers shared across MAFw.

class mafw.models.**FilterSchema**(*root_model: type[~mafw.db.db_model.MAFwBaseModel], allowed_models: tuple[type[~mafw.db.db_model.MAFwBaseModel], ...] = <factory>*)

Bases: `object`

Static descriptor of the models available to the GUI filter builder.

Parameters

- **root_model** – Model that controls the primary query entry point.
- **allowed_models** – Supplementary models that can be joined during filtering.

class mafw.models.**LoopItem**(*i_item: int*, *n_item: int*, *payload: Any*)

Bases: `object`

Container for a loop item and its position.

i_item: int

Index of the item within the loop.

n_item: int

Total number of items in the loop.

payload: Any

The item payload.

class mafw.models.**LoopResult**(*i_item: int*, *n_item: int*, *looping_status:* LoopingStatus, *payload: Any*, *duration: float*)

Bases: `object`

Container for a loop result and its status.

duration: float

Wall clock duration of the item processing.

i_item: int

Index of the item within the loop.

looping_status: *LoopingStatus*

Looping status produced while processing the item.

n_item: int

Total number of items in the loop.

payload: Any

Optional result payload returned by the processor.

class mafw.models.**ParameterSchema**(*name: str*, *annotation: type | None*, *default: Any*, *help: str | None*, *is_list: bool*, *is_dict: bool*)

Bases: `object`

Static metadata describing a single processor parameter.

Parameters

- **name** – Symbolic name used in steering files.
- **annotation** – Optional type hint supplied by the processor definition.
- **default** – The default value declared on the descriptor.
- **help** – Documentation text that should be surfaced to the user.

- **is_list** – True if the parameter expects a `list`-like value.
- **is_dict** – True if the parameter expects a `dict`-like value.

class mafw.models.**ProcessorSchema**(*parameters: list[ParameterSchema]*, *filter:* FilterSchema | *None*)

Bases: `object`

Static metadata describing a processor.

Parameters

- **parameters** – List of parameter schemas.
- **filter** – Optional filter schema.

get_parameter(*parameter_name: str*) → *ParameterSchema*

Return the parameter schema for the given parameter name.

get_parameter_names() → list[str]

Return the list of parameter names.

Modules

filter_schema	Static metadata describing the models available for filtering.
loop_payloads	Loop payload dataclasses for parallel processors.
parameter_schema	Schemas describing processor parameters.
processor_schema	Schemas describing a processor.

mafw.models.filter_schema

Static metadata describing the models available for filtering.

Classes

FilterSchema(root_model, allowed_models, ...] =)	Static descriptor of the models available to the GUI filter builder.

class mafw.models.filter_schema.**FilterSchema**(*root_model: type[~mafw.db.db_model.MAFwBaseModel]*, *allowed_models: tuple[type[~mafw.db.db_model.MAFwBaseModel], ...] = <factory>*)

Bases: `object`

Static descriptor of the models available to the GUI filter builder.

Parameters

- **root_model** – Model that controls the primary query entry point.
- **allowed_models** – Supplementary models that can be joined during filtering.

mafw.models.loop_payloads

Loop payload dataclasses for parallel processors.

Added in version v2.1.0.

Classes

LoopItem(i_item, n_item, payload)	Container for a loop item and its position.
LoopResult(i_item, n_item, looping_status, ...)	Container for a loop result and its status.

class mafw.models.loop_payloads.**LoopItem**(*i_item: int*, *n_item: int*, *payload: Any*)

Bases: object

Container for a loop item and its position.

i_item: int

Index of the item within the loop.

n_item: int

Total number of items in the loop.

payload: Any

The item payload.

class mafw.models.loop_payloads.**LoopResult**(*i_item: int*, *n_item: int*, *looping_status:* LoopingStatus, *payload: Any*, *duration: float*)

Bases: object

Container for a loop result and its status.

duration: float

Wall clock duration of the item processing.

i_item: int

Index of the item within the loop.

looping_status: *LoopingStatus*

Looping status produced while processing the item.

n_item: int

Total number of items in the loop.

payload: Any

Optional result payload returned by the processor.

mafw.models.parameter_schema

Schemas describing processor parameters.

These light-weight dataclasses capture the static metadata that is declared through *processor.ActiveParameter* definitions and can be consumed by tooling without instantiating the processor itself.

Classes

ParameterSchema(name, annotation, default, ...)	Static metadata describing a single processor parameter.

class mafw.models.parameter_schema.**ParameterSchema**(*name: str*, *annotation: type | None*, *default: Any*, *help: str | None*, *is_list: bool*, *is_dict: bool*)

Bases: object

Static metadata describing a single processor parameter.

Parameters

- **name** – Symbolic name used in steering files.
- **annotation** – Optional type hint supplied by the processor definition.
- **default** – The default value declared on the descriptor.
- **help** – Documentation text that should be surfaced to the user.
- **is_list** – True if the parameter expects a `list`-like value.
- **is_dict** – True if the parameter expects a `dict`-like value.

mafw.models.processor_schema

Schemas describing a processor.

These light-weight dataclasses capture the static metadata that is declared through *base.Processor* definitions and can be consumed by tooling without instantiating the processor itself.

Classes

ProcessorSchema(parameters, filter)	Static metadata describing a processor.

class mafw.models.processor_schema.**ProcessorSchema**(*parameters: list[*ParameterSchema*]*, *filter:* FilterSchema | *None*)

Bases: `object`

Static metadata describing a processor.

Parameters

- **parameters** – List of parameter schemas.
- **filter** – Optional filter schema.

get_parameter(*parameter_name: str*) → *ParameterSchema*

Return the parameter schema for the given parameter name.

get_parameter_names() → list[str]

Return the list of parameter names.

14.1.11 mafw.plugin_manager

Plugin management system for MAFw framework.

This module provides the core functionality for loading and managing plugins within the MAFw framework. It supports loading various types of plugins including processors, standard tables, database models, and user interfaces.

The plugin manager uses the pluggy library to handle plugin discovery and registration through entry points and hooks.

When plugins are loaded using the *MAFwPluginManager.load_plugins()* function, the job is divided into multiple threads to improve performance.

Key features:

- Asynchronous plugin loading with progress indication
- Support for both internal and external plugins
- Type-safe plugin handling with proper data structures
- Logging integration for monitoring plugin loading processes
- Global plugin manager singleton for consistent access

The module defines several key components:

- *LoadedPlugins*: Data container for loaded plugins
- *MAFwPluginManager*: Main plugin manager class
- *get_plugin_manager()*: Factory function for accessing the plugin manager

Plugin types supported:

- Processors (*processors*): Classes that implement data processing logic
- Database Modules (*db_modules*): Model modules for database interaction
- User Interfaces (*ui*): UI implementations for different interfaces

Changed in version v2.0.0: Complete refactoring of the plugin manager system.

Module Attributes

PluginTypes	Type alias for accepted types of plugins.
global_mafw_plugin_manager	The global mafw plugin manager dictionary.

Functions

get_plugin_manager([force_recreate])	Create a new or return an existing plugin manager for a given project

Classes

LoadedPlugins(processor_list, ...)	Container class for storing loaded plugins of various types.
MAFwPluginManager([project_name])	The MAFw plugin manager.

class mafw.plugin_manager.**LoadedPlugins**(*processor_list: list[~mafw.lazy_import.ProcessorClassProtocol] = <factory>, processor_dict: dict[str, ~mafw.lazy_import.ProcessorClassProtocol] = <factory>, db_model_modules: list[str] = <factory>, ui_list: list[~mafw.lazy_import.UserInterfaceClassProtocol] = <factory>, ui_dict: dict[str, ~mafw.lazy_import.UserInterfaceClassProtocol] = <factory>*)

Bases: `object`

Container class for storing loaded plugins of various types.

This dataclass holds collections of different plugin types that have been loaded by the *MAFwPluginManager*. It provides organized storage for processors, database modules, and user interfaces.

Added in version v2.0.0.

db_model_modules: list[str]

List of database model module names.

processor_dict: dict[str, *ProcessorClassProtocol*]

Dictionary mapping processor names to their classes.

processor_list: list[*ProcessorClassProtocol*]

List of loaded processor classes.

ui_dict: dict[str, *UserInterfaceClassProtocol*]

Dictionary mapping user interface names to their classes.

ui_list: list[*UserInterfaceClassProtocol*]

List of loaded user interface classes.

class mafw.plugin_manager.**MAFwPluginManager**(*project_name: str = 'mafw'*)

Bases: PluginManager

The MAFw plugin manager.

The MAFwPluginManager class manages the loading and registration of plugins within the MAFw framework. It supports asynchronous loading of various plugin types, including processors, database modules, and user interfaces, using a thread pool executor for improved performance.

The class provides methods to load each type of plugin and handles delayed status messages if loading takes longer than expected.

Added in version v2.0.0.

_delayed_status_message(*futures: list[Future[Any]]*) → None

Display a warning message if plugin loading takes longer than expected.

This method is called after a delay to check if all plugin loading operations have completed. If not, it logs a warning message to inform the user that plugin loading is taking longer than expected.

Parameters
futures (*list[concurrent.futures.Future]*) – List of futures representing ongoing plugin loading operations

load_db_models_plugins() → list[str]

Load database model modules from the plugin manager.

This method retrieves all database model modules registered through the plugin manager's *register_db_model_modules()* hook and imports them.

Returns
List of database model module names

Return type
list[str]

load_plugins(*plugins_to_load: Iterable[Literal['processors', 'db_modules', 'ui']]*) → *LoadedPlugins*

Load plugins of specified types in multiple threads.

This method loads plugins of the specified types using a thread pool executor for improved performance. It handles different plugin types including processors, standard tables, database modules, and user interfaces.

Parameters
plugins_to_load (Iterable[*PluginTypes*]) – Iterable of plugin types to load

Returns
Container with loaded plugins of all requested types

Return type
LoadedPlugins

load_processor_plugins() → tuple[list[*ProcessorClassProtocol*], dict[str, *ProcessorClassProtocol*]]

Load available processor plugins from the plugin manager.

This method retrieves all processor plugins registered through the plugin manager's *register_processors()* hook. *register_processors()* hook.

Returns
A tuple containing: - List of available processor classes - Dictionary mapping processor names to their classes

Return type
tuple[list[type[*processor.Processor*]], dict[str, type[*processor.Processor*]]]

load_user_interface_plugins() → tuple[list[*UserInterfaceClassProtocol*], dict[str, *UserInterfaceClassProtocol*]]

Load available user interface plugins from the plugin manager.

This method retrieves all user interface plugins registered through the plugin manager's *register_user_interfaces()* hook.

Returns
A tuple containing: - List of available user interface classes - Dictionary mapping user interface names to their classes

Return type
tuple[list[type[*UserInterfaceBase*]], dict[str, type[*UserInterfaceBase*]]]

max_loading_delay = 1

Loading delay before displaying a message.

If the loading of the external plugins is taking more than this value, a message is displayed to inform the user.

mafw.plugin_manager.**_as_db_module_result**(*obj: object*) → list[str]

Cast an object to the expected database module result type.

This helper function is used to convert the raw result from plugin loading operations into the expected list format for database modules.

Added in version v2.0.0.

Parameters
obj (*object*) – The object to cast

Returns
A list of database module names

Return type
list[str]

mafw.plugin_manager.**_as_processor_result**(*obj: object*) → tuple[list[*ProcessorClassProtocol*], dict[str, *ProcessorClassProtocol*]]

Cast an object to the expected processor result type.

This helper function is used to convert the raw result from plugin loading operations into the expected tuple format for processors.

Added in version v2.0.0.

Parameters
obj (*object*) – The object to cast

Returns
A tuple containing a list of processor classes and a dictionary mapping processor names to their classes

Return type
tuple[list[*ProcessorClassProtocol*], dict[str, *ProcessorClassProtocol*]]

mafw.plugin_manager.**_as_ui_result**(*obj: object*) → tuple[list[*UserInterfaceClassProtocol*], dict[str, *UserInterfaceClassProtocol*]]

Cast an object to the expected UI result type.

This helper function is used to convert the raw result from plugin loading operations into the expected tuple format for user interfaces.

Added in version v2.0.0.

Parameters
obj (*object*) – The object to cast

Returns
A tuple containing a list of UI classes and a dictionary mapping UI names to their classes

Return type
tuple[list[*UserInterfaceClassProtocol*], dict[str, *UserInterfaceClassProtocol*]]

mafw.plugin_manager.**get_plugin_manager**(*force_recreate: bool = False*) → *MAFwPluginManager*

Create a new or return an existing plugin manager for a given project

Parameters
force_recreate (*bool, Optional*) – Flag to force the creation of a new plugin manager. Defaults to False

Returns
The plugin manager

Return type
pluggy.PluginManager

mafw.plugin_manager.**PluginTypes**

Type alias for accepted types of plugins.

alias of `Literal['processors', 'db_modules', 'ui']`

mafw.plugin_manager.**global_mafw_plugin_manager: dict[str, *MAFwPluginManager*] = {}**

The global mafw plugin manager dictionary.

14.1.12 mafw.plugins

Exports Processor classes to the execution script.

Functions

register_db_model_modules()	Returns the list of modules with the database model definitions
register_processors()	Returns a list of processors to be registered
register_user_interfaces()	Returns a list of user interfaces that can be used

mafw.plugins.**register_db_model_modules**() → list[str]

Returns the list of modules with the database model definitions

mafw.plugins.**register_processors**() → list[*ProcessorClassProtocol*]

Returns a list of processors to be registered

mafw.plugins.**register_user_interfaces**() → list[*UserInterfaceClassProtocol*]

Returns a list of user interfaces that can be used

14.1.13 mafw.processor

Processor package — core building block of the MAFw pipeline.

This package provides the *Processor* base class, the *ProcessorList* container, parameter descriptors, and supporting utilities.

Changed in version 2.3: Refactored from a single-file module into a multi-module package for improved maintainability. All public imports remain unchanged.

class mafw.processor.**ActiveParameter**(*name: str*, *value:* ParameterType | *None* = *None*, *default:* ParameterType | *None* = *None*, *help_doc: str* = '')

Bases: Generic[*ParameterType*]

The public interface to the processor parameter.

The behaviour of a *processor.Processor* can be customised by using processor parameters. The value of these parameters can be either set via a configuration file or directly when creating the class.

If the user wants to benefit from this facility, they have to add in the instance of the Processor subclass an ActiveParameter instance in this way:

```
class MyProcessor(Processor):

    # this is the input folder
    input_folder = ActiveParameter('input_folder', Path(r'C:\'), help_doc='This␣
↪is where to look for input files')

    def __init__(self, *args, **kwargs):
        super().__init(*args, **kwargs)

        # change the input folder to something else
        self.input_folder = Path(r'D:\data')

        # get the value of the parameter
        print(self.input_folder)
```

The ActiveParameter is a descriptor, it means that when you create one of them, a lot of work is done behind the scene.

In simple words, a processor parameter is made by two objects: a public interface where the user can easily access the value of the parameter and a private interface where all other information (default, documentation…) is also stored.

The user does not have to take care of all of this. When a new ActiveParameter instance is added to the class as in the code snippet above, the private interface is automatically created and will stay in the class instance until the end of the class lifetime.

To access the private interface, the user can use the *processor.Processor.get_parameter()* method using the parameter name as a key.

The user can assign to an ActiveParameter almost any name. There are just a few invalid parameter names that are used for other purposes. The list of reserved names is available *here*. Should the user inadvertently use a reserved named, a *ProcessorParameterError* is raised.

See also

The private counterpart in the *processor.PassiveParameter*.

An explanation on how processor parameters work and should be used is given in *Understanding processor parameters*

The list of *reserved names*.

Constructor parameters:

Parameters

- **name** (*str*) – The name of the parameter.
- **value** (ParameterType, *Optional*) – The initial value of the parameter. Defaults to None.

- **default** (*ParameterType, Optional*) – The default value of the parameter, to be used when `value` is not set., Defaults to None.
- **help_doc** (*str, Optional*) – An explanatory text describing the parameter.

static _matches_container(*annotation: type | None*, *default_value: Any*, *container: type*) → bool

Generic check for whether annotation/default matches a given container type.

_is_dict_annotation(*annotation: type | None*, *default_value: Any*) → bool

Check whether the annotation or default value indicates a dict type.

_is_list_annotation(*annotation: type | None*, *default_value: Any*) → bool

Check whether the annotation or default value indicates a list type.

_resolve_parameter_annotation() → type | None

Resolve the type annotation for this parameter from the owner class hints.

_schema_default_value() → Any

Return a safe copy of the default/initial value for schema reporting.

_validate_name(*proposed_name: str*) → str

Validate that the proposed parameter name is not in the list of forbidden names.

This private method checks if the provided name is allowed for use as a processor parameter. Names that are listed in *reserved_names* cannot be used as parameter names.

Parameters
proposed_name (*str*) – The name to be validated for use as a processor parameter.

Returns
The validated name if it passes the forbidden names check.

Return type
str

Raises
ProcessorParameterError – If the proposed name is in the list of forbidden names.

to_schema() → *ParameterSchema*

Returns the static schema describing this parameter.

The schema is derived solely from the descriptor metadata and does not instantiate the owning processor.

reserved_names: list[str] = ['__logic__', '__filter__', '__new_only__', '__inheritance__', '__enable__']

A list of names that cannot be used as processor parameter names.

- *__logic__*
- *__filter__*
- *__new_only__*
- *__inheritance__*
- *__enable__*

class mafw.processor.**PassiveParameter**(*name: str*, *value:* ParameterType | *None = None*, *default:* ParameterType | *None = None*, *help_doc: str = ''*)

Bases: Generic[*ParameterType*]

An helper class to store processor parameter value and metadata.

This class is the private interface used by the *ActiveParameter* descriptor to store its value and metadata.

When a new *ActiveParameter* is added to a class, an instance of a PassiveParameter is added to the processor parameter *register*.

> **See also**
>
> An explanation on how processor parameters work and should be used is given in *Understanding processor parameters*

Changed in version v2.0.0: User should only use *ActiveParameter* and never manually instantiate *PassiveParameter*.

Constructor parameters:

Parameters

- **name** (*str*) – The name of the parameter. It must be a valid python identifier.
- **value** (*ParameterType, Optional*) – The set value of the parameter. If None, then the default value will be used. Defaults to None.
- **default** (*ParameterType, Optional*) – The default value for the parameter. It is used if the *value* is not provided. Defaults to None.
- **help_doc** (*str, Optional*) – A brief explanation of the parameter.

Raises

ProcessorParameterError – if both *value* and *default* are not provided or if *name* is not a valid identifier.

property is_optional: bool

Property to check if the parameter is optional.

Returns

True if the parameter is optional

Return type

bool

property is_set: bool

Property to check if the value has been set.

It is useful for optional parameter to see if the current value is the default one, or if the user set it.

property value: *ParameterType*

Gets the parameter value.

Returns

The parameter value.

Return type

ParameterType

Raises

ProcessorParameterError – if both value and default were not defined.

class mafw.processor.**Processor**(**args: Any*, ***kwargs: Any*)

Bases: object

The basic processor.

A very comprehensive description of what a Processor does and how it works is available at *Processor: The core of MAFw*.

Constructor parameters

Parameters

- **name** (*str, Optional*) – The name of the processor. If None is provided, the class name is used instead. Defaults to None.

- **description** (*str, Optional*) – A short description of the processor task. Defaults to the processor name.
- **config** (*dict, Optional*) – A configuration dictionary for this processor. Defaults to None.
- **looper** (LoopType, *Optional*) – Enumerator to define the looping type. Defaults to LoopType.ForLoop
- **user_interface** (UserInterfaceBase, *Optional*) – A user interface instance to be used by the processor to interact with the user.
- **timer** (Timer, *Optional*) – A timer object to measure process duration.
- **timer_params** (*dict, Optional*) – Parameters for the timer object.
- **database** (*Database, Optional*) – A database instance. Defaults to None.
- **database_conf** (*dict, Optional*) – Configuration for the database. Default to None.
- **remove_orphan_files** (*bool, Optional*) – Boolean flag to remove files on disc without a reference to the database. See *Standard tables* and *_remove_orphan_files()*. Defaults to True
- **replica_id** (*str, Optional*) – The replica identifier for the current processor.
- **create_standard_tables** (*bool, Optional*) – Boolean flag to create std tables on disk. Defaults to True When a nested steering configuration is loaded, this value can be overridden by the global create_standard_tables entry. Flat processor configurations keep the constructor value.
- **max_workers** (*int, Optional*) – Number of worker threads for parallel loops.
- **queue_size** (*int, Optional*) – Maximum size of the internal queue for the queue-based parallel loop.
- **queue_batch_size** (*int, Optional*) – Number of items processed per worker task in the queue-based parallel loop.
- **kwargs** – Keyword arguments that can be used to set processor parameters.

classmethod _apply_super_call_wrappers() → None

Wraps overridden methods so the class can detect whether they called *super()*.

This method runs immediately after the class is created (see *processor.ProcessorMeta*). For every method that we expect scientists to extend (start, finish, etc.) we replace their implementation with a wrapper. The wrapper resets a per-instance flag, invokes the real override, and only after that method returns it checks whether the real *super()* was ever reached; if not, it emits a *MissingSuperCall* warning. In other words, the wiring happens while the subclass is defined, and the actual smoke test executes each time the method runs.

classmethod filter_schema() → *FilterSchema* | None

Optional metadata describing the models available for filtering.

classmethod parameter_schema() → list[*ParameterSchema*]

Return the ordered static schema for the processor parameters defined on the class.

The schema is derived without instantiating the processor, keeping toolchains free from side effects.

classmethod processor_schema() → *ProcessorSchema*

Return the static schema for the processor.

static _compute_default_max_workers() → int

Helper to compute the default number of workers for parallel execution.

Returns min(32, cpu_count + 4).

_build_loop_item() → *LoopItem*

Build a LoopItem payload for the current loop context.

Returns
The LoopItem payload.

Return type
mafw.models.LoopItem

_build_loop_result(*payload: Any*, *duration: float*) → *LoopResult*

Build a LoopResult payload for the current loop context.

Parameters

- **payload** (*Any*) – The optional payload returned by process.
- **duration** (*float*) – Wall-clock duration of the item processing.

Returns
The LoopResult payload.

Return type
mafw.models.LoopResult

_call_with_optional_payload(*func: Callable[[...], Any]*, *payload: Any*) → Any

Invoke a processor hook with an optional payload if the signature allows it.

Parameters

- **func** (*Callable*) – The callable to invoke.
- **payload** (*Any*) – The payload to pass if supported.

Returns
The callable return value.

Return type
Any

_check_method_overload() → None

Check if the user overloaded the required methods.

Depending on the loop type, the user must overload different methods. This method is doing the check and if the required methods are not overloaded a warning is emitted.

_did_call_super(*method: str*) → bool

Check whether a method called its super implementation.

_execute_for_loop() → None

Executes the processor within a for loop.

Private method. Do not overload nor invoke it directly. The *execute()* method will call the appropriate implementation depending on the processor LoopType.

_execute_single() → None

Execute the processor in single mode.

Private method. Do not overload nor invoke it directly. The *execute()* method will call the appropriate implementation depending on the processor LoopType.

_execute_while_loop() → None

Executes the processor within a while loop.

Private method. Do not overload nor invoke it directly. The *execute()* method will call the appropriate implementation depending on the processor LoopType.

_in_thread_context() → bool

Return True when running inside a parallel worker thread.

_load_parameter_configuration() → None

Load processor parameter configuration from the internal configuration dictionary.

This method processes the processor's configuration dictionary to set parameter values. It handles two configuration formats:

1. Nested format: `{'ProcessorName': {'param1': value1, ...}}`
2. Flat format: `{'param1': value1, ...}`

The method also handles filter configurations by collecting filter table names and deferring their initialisation until after the global filter has been processed.

Changed in version v2.0.0: For option 1 combining configuration from name and name_replica

Changed in version v2.1.2: When a nested steering configuration is loaded, the processor-level create_standard_tables value is overridden from the global steering-file setting. Flat configurations keep the constructor value untouched.

Raises
ProcessorParameterError – If a parameter in the configuration is not registered.

See also

mafw.db.db_filter.ModelFilter.from_conf()

_mark_super_call(*method: str*) → None

Mark a method as having called its super implementation.

_override_defaults() → None

Override default parameter values with values from *new_defaults*.

This private method iterates through the *new_defaults* dictionary and updates the corresponding processor parameters with new values. Only parameters that exist in both *new_defaults* and *_processor_parameters* are updated.

Added in version v2.0.0.

_overrule_kws_parameters() → None

Override processor parameters with values from keyword arguments.

This method applies parameter values passed as keyword arguments during processor initialisation. It ensures that the parameter types match the expected types before setting the values.

See also

_register_parameters(), *_load_parameter_configuration()*, *set_parameter_value()*

_payload_annotation_matches(*annotation: Any*) → bool

Check whether an annotation matches LoopItem or LoopResult.

Handles both `typing.Union[X, Y]` and the built-in `X | Y` union syntax.

Parameters
annotation (*Any*) – The annotation to inspect.

Returns
True if the annotation matches LoopItem or LoopResult.

Return type
bool

_process_for_loop(*item_list: Collection[Any]*) → None

Process items with the standard serial for-loop.

Parameters
item_list (*Collection[Any]*) – The list of items to process.

_process_parallel_for_loop(*item_list: Collection[Any]*) → None

Process items in parallel using a thread pool.

Parameters
item_list (*Collection[Any]*) – The list of items to process.

_process_parallel_for_loop_with_queue(*item_list: Collection[Any]*) → None

Process items in parallel using a producer/consumer queue.

Items are processed in worker threads, while a dedicated consumer thread handles the post-processing hooks.

Parameters
item_list (*Collection[Any]*) – The list of items to process.

_register_parameters() → None

Register processor parameters defined as ActiveParameter instances in the class.

This private method scans the class definition for any *processor.ActiveParameter* instances and creates corresponding *processor.PassiveParameter* instances to store the actual parameter values and metadata. It ensures that all processor parameters are properly initialised and available for configuration through the processor's configuration system.

The method checks for duplicate parameter names and raises a *ProcessorParameterError* if duplicates are detected. It also sets the internal flag *processor.Processor._parameter_registered* to True once registration is complete.

Note

This method is automatically called during processor initialisation and should not be called directly by users.

See also

processor.Processor, *processor.Processor._override_defaults()*, *processor.Processor._load_parameter_configuration()*, *processor.Processor._overrule_kws_parameters()*

Changed in version v2.0.0: Only *processor.ActiveParameter* are not registered. The use of *processor.PassiveParameter* is only meant to store the value and metadata of the active counterpart.

_remove_orphan_files() → None

Remove orphan files.

If a connection to the database is available, then the OrphanFile standard table is queried for all its entries, and all the files are then removed.

The user can turn off this behaviour by switching the *remove_orphan_files* to False.

_reset_parameters() → None

Reset processor parameters to their initial state.

This method clears all currently registered processor parameters and triggers a fresh registration process. It's useful when parameter configurations need to be reinitialized or when parameters have been modified and need to be reset.

See also

_register_parameters(), *_register_parameters()*

_reset_super_call_flag(*method: str*) → None

Reset the super-call flag for a method.

_thread_loop_context(*i_item: int*, *n_item: int*, *item: Any*) → Iterator[None]

Context manager to set thread-local loop attributes for parallel execution.

Parameters

- **i_item** (*int*) – Item index for the loop.
- **n_item** (*int*) – Total number of items in the loop.
- **item** (*Any*) – Item payload.

accept_item() → None

Does post process actions on a successfully processed item.

Within the *process()*, the user left the looping status to Continue, so it means that everything looks good and this is the right place to perform database updates or file savings.

In parallel for loops, the method can optionally accept a *LoopResult* parameter for direct access to the processed payload and looping status.

acquire_resources() → None

Acquires resources and add them to the resource stack.

The whole body of the *execute()* method is within a context structure. The idea is that if any part of the code inside should throw an exception that breaking the execution, we want to be sure that all stateful resources are properly closed.

Since the number of resources may vary, the variable number of nested *with* statements has been replaced by an ExitStack. Resources, like open files, timers, db connections, need to be added to the resource stacks in this method.

In the case a processor is being executed within a *ProcessorList*, then some resources might be shared, and for this reason they are not added to the stack. This selection can be done via the private *local_resource_acquisition*. This is normally True, meaning that the processor will handle its resources independently, but when the processor is executed from a *ProcessorList*, this flag is automatically turned to False.

If the user wants to add additional resources, he has to overload this method calling the super to preserve the original resources. If he wants to have shared resources among different processors executed from inside a processor list, he has to overload the *ProcessorList* class as well.

consumer_finish() → None

Executes once in the consumer thread after the queue has been drained.

consumer_process(*loop_result:* LoopResult | *None* = *None*) → None

Handle processed items in the consumer thread for the parallel queue loop.

By default this dispatches to *accept_item()* or *skip_item()* based on the looping status. The method can optionally accept a *LoopResult* payload.

consumer_start() → None

Executes once in the consumer thread for the parallel queue loop.

This hook mirrors *start()* but is only used with the queue-based parallel loop.

delete_parameter(*name: str*) → None

Deletes a processor parameter.

Parameters
name (*str*) – The name of the parameter to be deleted.

Raises
ProcessorParameterError – If a parameter with *name* is not registered.

dump_parameter_configuration(*option: int = 1*) → dict[str, Any]

Dumps the processor parameter values in a dictionary.

The snippet below explains the meaning of *option*.

```
# option 1
conf_dict1 = {
    'Processor': {'param1': 5, 'input_table': 'my_table'}
}

# option 2
conf_dict2 = {'param1': 5, 'input_table': 'my_table'}
```

In the case of option 1, the replica aware name (*replica_name()*) will be used as a key for the configuration dictionary.

Changed in version v2.0.0: With option 1, using *replica_name()* instead of *name* as key of the configuration dictionary.

Parameters
option (*int, Optional*) – Select the dictionary style. Defaults to 1.

Returns
A parameter configuration dictionary.

Return type
dict

execute() → None

Execute the processor tasks.

This method works as a dispatcher, reassigning the call to a more specific execution implementation depending on the *loop_type*.

finish() → None

Concludes the execution.

The user can reimplement this method if there are some conclusive tasks that must be achieved. Always include a call to super().

format_progress_message() → None

Customizes the progress message with information about the current item.

The user can overload this method in order to modify the message being displayed during the process loop with information about the current item.

The user can access the current value, its position in the looping cycle and the total number of items using *processor.Processor.item*, *processor.Processor.i_item* and *processor.Processor.n_item*.

get_filter(*model_name: str*) → *ModelFilter*

Returns a registered `ModelFilter` via the model name.

If a filter for the provided model_name does not exist, a KeyError is raised.

Parameters
model_name (*str*) – The model name for which the filter will be returned.

Returns
The registered filter

Return type
mafw.db.db_filter.ModelFilter

Raises
KeyError is a filter with the give name is not found.

get_items() → Collection[Any]

Returns the item collections for the processor loop.

This method must be overloaded for the processor to work. Generally, this is getting a list of rows from the database, or a list of files from the disk to be processed.

Returns
A collection of items for the loop

Return type
Collection[Any]

get_parameter(*name: str*) → *PassiveParameter*[Any]

Gets the processor parameter named name.

Parameters
name (*str*) – The name of the parameter.

Returns
The processor parameter

Return type
processor.PassiveParameter

Raises
ProcessorParameterError – If a parameter with *name* is not registered.

get_parameters() → dict[str, *PassiveParameter*[Any]]

Returns the full dictionary of registered parameters for this processor.

Useful when dumping the parameter specification in a configuration file, for example.

Returns
The dictionary with the registered parameters.

Return type
dict[str, *processor.PassiveParameter*[*ParameterType*]

initialise_parameters() → None

Initialises processor parameters by registering them and applying various configuration sources.

This method orchestrates the parameter initialisation process by performing the following steps in order:

1. Registers processor parameters defined as `processor.ActiveParameter` instances
2. Overrides default parameter values with any configured overrides
3. Loads parameter configuration from the processor's configuration dictionary
4. Applies keyword arguments as parameter overrides

The method ensures that all processor parameters are properly configured before the processor execution begins. It is automatically called during processor initialisation and should not typically be called directly by users.

See also

_register_parameters(), *_override_defaults()*, *_load_parameter_configuration()*, *_overrule_kws_parameters()*

Added in version v2.0.0.

on_looping_status_set(*status:* LoopingStatus) → None

Call back invoked when the looping status is set.

The user can overload this method according to the needs.

Parameters

status (LoopingStatus) – The set looping status.

on_processor_status_change(*old_status:* ProcessorStatus, *new_status:* ProcessorStatus) → None

Callback invoked when the processor status is changed.

Parameters

- **old_status** (ProcessorStatus) – The old processor status.
- **new_status** (ProcessorStatus) – The new processor status.

print_process_statistics() → None

Print the process statistics.

A utility method to display the fastest, the slowest and the average timing required to process on a single item. This is particularly useful when the looping processor is part of a ProcessorList.

process() → None

Processes the current item.

This is the core of the Processor, where the user has to define the calculations required.

In parallel for loops, the method can optionally accept a *LoopItem* parameter if the user prefers not to rely on thread-local access to *processor.Processor.item*, *processor.Processor.i_item* and *processor.Processor.n_item*.

set_parameter_value(*name: str*, *value:* ParameterType) → None

Sets the value of a processor parameter.

Parameters

- **name** (*str*) – The name of the parameter to be deleted.
- **value** (ParameterType) – The value to be assigned to the parameter.

Raises

ProcessorParameterError – If a parameter with *name* is not registered.

skip_item() → None

Does post process actions on a *NOT* successfully processed item.

Within the *process()*, the user set the looping status to Skip, so it means that something went wrong and here corrective actions can be taken if needed.

In parallel for loops, the method can optionally accept a *LoopResult* parameter for direct access to the processed payload and looping status.

start() → None

Start method.

The user can overload this method, including all steps that should be performed at the beginning of the operation.

If the user decides to overload it, it should include a call to the super method.

validate_configuration() → None

Validate the configuration provided via the processor parameters.

Method to be implemented by subclasses if a configuration validation is needed.

The method should silently check for the proper configuration, if this is not obtained, then the *InvalidConfigurationError* must be raised.

Added in version v2.0.0.

while_condition() → bool

Return the while condition

Returns
True if the while loop has to continue, false otherwise.

Return type
bool

_config: dict[str, Any]

A dictionary containing the processor configuration object.

This dictionary is populated with configuration parameter (always type 2) during the *_load_parameter_configuration()* method.

The original value of the configuration dictionary that is passed to the constructor is stored in *_orig_config*.

Changed in version v2.0.0: Now it is an empty dictionary until the *_load_parameter_configuration()* is called.

_ids = count(0)

A counter for all processor instances

_item: Any

The current item of the loop.

_looping_status: *LoopingStatus*

The looping status for the main thread.

_methods_to_be_checked_for_super: tuple[str, ...] = ('start', 'finish')

List of methods that should invoke their super implementation when overridden.

_orig_config

A copy of the original configuration dictionary.

Added in version v2.0.0.

_parameter_registered

A boolean flag to confirm successful parameter registration.

_processor_parameters: OrderedDict[str, *PassiveParameter*[Any]]

A dictionary to store all the processor parameter instances.

The name of the parameter is used as a key, while for the value an instance of the *processor.PassiveParameter* is used.

_super_call_flags: dict[str, bool]

Tracks super-call usage for methods that require it.

_thread_local

Thread-local storage for loop attributes in parallel execution.

Added in version v2.1.0.

_wall_clock_start: float | None

Timestamp when the looping execution began.

create_standard_tables

The boolean flag to proceed or skip with standard table creation and initialisation

property database: Database

Returns the database instance

Returns
A database object.

Raises
MissingDatabase – If the database connection has not been established.

description

A short description of the processor task.

filter_register: *ProcessorFilter*

The DB filter register of the Processor.

property i_item: int

The enumeration of the current item being processed.

property item: Any

The current item of the loop.

property local_resource_acquisition: bool

Checks if resources should be acquired locally.

When the processor is executed in stand-alone mode, it is responsible to acquire and release its own external resources, but when it is executed from a ProcessorList, then is a good practice to share and distribute resources among the whole processor list. In this case, resources should not be acquired locally by the single processor, but from the parent execution context.

Returns
True if resources are to be acquired locally by the processor. False, otherwise.

Return type
bool

loop_type: *LoopType*

The loop type.

The value of this parameter can also be changed by the *execution_workflow()* decorator factory.

See *LoopType* for more details.

property looping_status: *LoopingStatus*

The looping status for the current thread context.

max_workers

Maximum number of worker threads used in parallel loops.

property n_item: int | None

The total number of items to be processed or None for an undefined loop

name

The name of the processor.

new_defaults: dict[str, Any] = {}

A dictionary containing defaults value for the parameters to be overridden

Added in version v2.0.

processor_exit_status

Processor exit status

processor_status

Processor execution status

progress_message: str = 'Processor is working'

Message displayed to show the progress.

It can be customized with information about the current item in the loop by overloading the *format_progress_message()*.

queue_batch_size

Number of items processed per worker task in *ParallelForLoopWithQueue*.

queue_size

Maximum size of the queue used by *ParallelForLoopWithQueue*.

remove_orphan_files: bool

The flag to remove or protect the orphan files. Defaults to True

replica_id

The replica identifier specified in the constructor

Added in version v2.0.0.

property replica_name: str

Returns the replica aware name of the processor.

If no replica_id is specified, then return the pure name, otherwise join the two string using the '#' symbol.

Added in version v2.0.0.

Returns
The replica aware name of the processor.

Return type
str

unique_id

A unique identifier representing how many instances of Processor has been created.

property unique_name: str

Returns the unique name for the processor.

class mafw.processor.**ProcessorList**(**args: Processor | ProcessorList, name: str | None = None, description: str | None = None, timer: Timer | None = None, timer_params: dict[str, Any] | None = None, user_interface: UserInterfaceBase | None = None, database: Database | None = None, database_conf: dict[str, Any] | None = None, create_standard_tables: bool = True*)

Bases: list[Processor | ProcessorList]

A list like collection of processors.

ProcessorList is a subclass of list containing only Processor subclasses or other ProcessorList.

An attempt to add an element that is not a Processor or a ProcessorList will raise a TypeError.

Along with an iterable of processors, a new processor list can be built using the following parameters.

Constructor parameters:

Parameters

- **name** (*str, Optional*) – The name of the processor list. Defaults to ProcessorList.
- **description** (*str, Optional*) – An optional short description. Default to ProcessorList.
- **timer** (*Timer, Optional*) – The timer object. If None is provided, a new one will be created. Defaults to None.
- **timer_params** (*dict, Optional*) – A dictionary of parameter to build the timer object. Defaults to None.
- **user_interface** (*UserInterfaceBase, Optional*) – A user interface. Defaults to None
- **database** (*Database, Optional*) – A database instance. Defaults to None.
- **database_conf** (*dict, Optional*) – Configuration for the database. Default to None.
- **create_standard_tables** (*bool, Optional*) – Whether or not to create the standard tables. Defaults to True.

static validate_item(*item:* Processor | ProcessorList) → *Processor* | *ProcessorList*

Validates the item being added.

static validate_items(*items: tuple[*Processor | ProcessorList*, ...] = ()*) → tuple[*Processor* | *ProcessorList*, ...]

Validates a tuple of items being added.

acquire_resources() → None

Acquires external resources.

The resource acquisition strategy mirrors *processor.Processor.acquire_resources()*: if a resource (timer, database) is already provided, it is reused; otherwise a new one is created and registered with the exit stack for automatic cleanup.

append(*_ProcessorList__object:* Processor | ProcessorList) → None

Appends a new processor at the end of the list.

distribute_resources(*processor:* Processor | *Self*) → None

Distributes the external resources to the items in the list.

execute() → *ProcessorExitStatus*

Execute the list of processors.

Similarly to the *processor.Processor*, ProcessorList can be executed. In simple words, the execute method of each processor in the list is called exactly in the same sequence as they were added.

extend(*_ProcessorList__iterable: Iterable[*Processor | ProcessorList*]*) → None

Extends the processor list with a list of processors.

insert(*_ProcessorList__index: SupportsIndex*, *_ProcessorList__object:* Processor | ProcessorList) → None

Adds a new processor at the specified index.

create_standard_tables

The boolean flag to proceed or skip with standard table creation and initialisation

property database: Database

Returns the database instance

Returns
A database instance

Raises
MissingDatabase – if a database connection is missing.

property name: str

The name of the processor list

Returns
The name of the processor list

Return type
str

nested_list

Boolean flag to identify that this list is actually inside another list.

Similarly to the local resource flag for the *base.Processor*, this flag prevent the user interface to be added to the resource stack.

property processor_exit_status: *ProcessorExitStatus*

The processor exit status.

It refers to the whole processor list execution.

class mafw.processor.**ProcessorMeta**(*name: str*, *bases: tuple[type, ...]*, *namespace: dict[str, Any]*)

Bases: type

Metaclass that finalizes Processor subclasses before instantiation.

Set up new Processor subclasses.

This method first checks the class for well-formed parameter definitions and filter metadata, then runs the super-call wrapping machinery described below. The wrappers are installed at class-creation time so that any later instance will be guarded before it even starts executing.

mafw.processor.**ensure_parameter_registration**(*func: _F*) → *_F*

Decorator to ensure that processor parameters are registered before *func* executes.

This is applied to methods that access `self._processor_parameters` and need the registration step to have completed first.

Parameters
func (_F) – The method to wrap.

Returns
The wrapped method.

Return type
_F

Raises
ProcessorParameterError – If applied to something that is not a Processor instance method.

mafw.processor.**validate_database_conf**(*database_conf: dict[str, Any] | None = None*) → dict[str, Any] | None

Validate the database configuration dictionary.

This function is called during processor initialisation to ensure that the supplied database configuration contains the minimum required fields (currently just `URL`). If the configuration is a full steering file, the `DBConfiguration` section is extracted automatically.

Parameters
database_conf (*dict, Optional*) – The input database configuration. Defaults to None.

Returns
Either the validated database configuration or None if it is invalid.

Return type
dict, None

Modules

base	Core Processor class for the MAFw pipeline.
meta	Metaclass for the Processor framework.
parameters	Parameter descriptors for the Processor framework.
processor_list	ProcessorList container for orchestrating sequences of processors.
utils	Standalone utility helpers for the Processor package.

mafw.processor.base

Core Processor class for the MAFw pipeline.

This module defines the *Processor* base class — the fundamental building block of every MAFw analysis pipeline. A Processor encapsulates a single processing step (reading data, applying corrections, fitting models, etc.) and can be orchestrated inside a *ProcessorList*.

Scientists subclass *Processor* and override a small set of methods (*get_items()*, *process()*, *accept_item()*) to define their analysis logic, while the framework manages looping, resource acquisition, parameter configuration, and progress reporting.

Added in version 2.3: Extracted from the monolithic `processor.py` module for improved maintainability and focused testing.

Classes

Processor(*args, **kwargs)	The basic processor.

class mafw.processor.base.**Processor**(**args: Any*, ***kwargs: Any*)

Bases: `object`

The basic processor.

A very comprehensive description of what a Processor does and how it works is available at *Processor: The core of MAFw*.

Constructor parameters

Parameters

- **name** (*str, Optional*) – The name of the processor. If None is provided, the class name is used instead. Defaults to None.
- **description** (*str, Optional*) – A short description of the processor task. Defaults to the processor name.
- **config** (*dict, Optional*) – A configuration dictionary for this processor. Defaults to None.
- **looper** (*LoopType, Optional*) – Enumerator to define the looping type. Defaults to LoopType.ForLoop
- **user_interface** (*UserInterfaceBase, Optional*) – A user interface instance to be used by the processor to interact with the user.

- **timer** (*Timer, Optional*) – A timer object to measure process duration.
- **timer_params** (*dict, Optional*) – Parameters for the timer object.
- **database** (*Database, Optional*) – A database instance. Defaults to None.
- **database_conf** (*dict, Optional*) – Configuration for the database. Default to None.
- **remove_orphan_files** (*bool, Optional*) – Boolean flag to remove files on disc without a reference to the database. See *Standard tables* and *_remove_orphan_files()*. Defaults to True
- **replica_id** (*str, Optional*) – The replica identifier for the current processor.
- **create_standard_tables** (*bool, Optional*) – Boolean flag to create std tables on disk. Defaults to True When a nested steering configuration is loaded, this value can be overridden by the global create_standard_tables entry. Flat processor configurations keep the constructor value.
- **max_workers** (*int, Optional*) – Number of worker threads for parallel loops.
- **queue_size** (*int, Optional*) – Maximum size of the internal queue for the queue-based parallel loop.
- **queue_batch_size** (*int, Optional*) – Number of items processed per worker task in the queue-based parallel loop.
- **kwargs** – Keyword arguments that can be used to set processor parameters.

classmethod _apply_super_call_wrappers() → None

Wraps overridden methods so the class can detect whether they called *super()*.

This method runs immediately after the class is created (see *processor.ProcessorMeta*). For every method that we expect scientists to extend (start, finish, etc.) we replace their implementation with a wrapper. The wrapper resets a per-instance flag, invokes the real override, and only after that method returns it checks whether the real *super()* was ever reached; if not, it emits a *MissingSuperCall* warning. In other words, the wiring happens while the subclass is defined, and the actual smoke test executes each time the method runs.

classmethod filter_schema() → *FilterSchema* | None

Optional metadata describing the models available for filtering.

classmethod parameter_schema() → list[*ParameterSchema*]

Return the ordered static schema for the processor parameters defined on the class.

The schema is derived without instantiating the processor, keeping toolchains free from side effects.

classmethod processor_schema() → *ProcessorSchema*

Return the static schema for the processor.

static _compute_default_max_workers() → int

Helper to compute the default number of workers for parallel execution.

Returns min(32, cpu_count + 4).

_build_loop_item() → *LoopItem*

Build a LoopItem payload for the current loop context.

Returns
The LoopItem payload.

Return type
mafw.models.LoopItem

_build_loop_result(*payload: Any*, *duration: float*) → *LoopResult*

Build a LoopResult payload for the current loop context.

Parameters

- **payload** (*Any*) – The optional payload returned by process.
- **duration** (*float*) – Wall-clock duration of the item processing.

Returns
The LoopResult payload.

Return type
mafw.models.LoopResult

_call_with_optional_payload(*func: Callable[[...], Any]*, *payload: Any*) → Any

Invoke a processor hook with an optional payload if the signature allows it.

Parameters

- **func** (*Callable*) – The callable to invoke.
- **payload** (*Any*) – The payload to pass if supported.

Returns
The callable return value.

Return type
Any

_check_method_overload() → None

Check if the user overloaded the required methods.

Depending on the loop type, the user must overload different methods. This method is doing the check and if the required methods are not overloaded a warning is emitted.

_did_call_super(*method: str*) → bool

Check whether a method called its super implementation.

_execute_for_loop() → None

Executes the processor within a for loop.

Private method. Do not overload nor invoke it directly. The *execute()* method will call the appropriate implementation depending on the processor LoopType.

_execute_single() → None

Execute the processor in single mode.

Private method. Do not overload nor invoke it directly. The *execute()* method will call the appropriate implementation depending on the processor LoopType.

_execute_while_loop() → None

Executes the processor within a while loop.

Private method. Do not overload nor invoke it directly. The *execute()* method will call the appropriate implementation depending on the processor LoopType.

_in_thread_context() → bool

Return True when running inside a parallel worker thread.

_load_parameter_configuration() → None

Load processor parameter configuration from the internal configuration dictionary.

This method processes the processor's configuration dictionary to set parameter values. It handles two configuration formats:

1. Nested format: `{'ProcessorName': {'param1': value1, ...}}`
2. Flat format: `{'param1': value1, ...}`

The method also handles filter configurations by collecting filter table names and deferring their initialisation until after the global filter has been processed.

Changed in version v2.0.0: For option 1 combining configuration from name and name_replica

Changed in version v2.1.2: When a nested steering configuration is loaded, the processor-level create_standard_tables value is overridden from the global steering-file setting. Flat configurations keep the constructor value untouched.

Raises
ProcessorParameterError – If a parameter in the configuration is not registered.

See also
mafw.db.db_filter.ModelFilter.from_conf()

_mark_super_call(*method: str*) → None

Mark a method as having called its super implementation.

_override_defaults() → None

Override default parameter values with values from *new_defaults*.

This private method iterates through the *new_defaults* dictionary and updates the corresponding processor parameters with new values. Only parameters that exist in both *new_defaults* and *_processor_parameters* are updated.

Added in version v2.0.0.

_overrule_kws_parameters() → None

Override processor parameters with values from keyword arguments.

This method applies parameter values passed as keyword arguments during processor initialisation. It ensures that the parameter types match the expected types before setting the values.

See also
_register_parameters(), *_load_parameter_configuration()*, *set_parameter_value()*

_payload_annotation_matches(*annotation: Any*) → bool

Check whether an annotation matches LoopItem or LoopResult.

Handles both `typing.Union[X, Y]` and the built-in `X | Y` union syntax.

Parameters
annotation (*Any*) – The annotation to inspect.

Returns
True if the annotation matches LoopItem or LoopResult.

Return type
bool

_process_for_loop(*item_list: Collection[Any]*) → None

Process items with the standard serial for-loop.

Parameters
item_list (*Collection[Any]*) – The list of items to process.

_process_parallel_for_loop(*item_list: Collection[Any]*) → None

Process items in parallel using a thread pool.

Parameters
item_list (*Collection[Any]*) – The list of items to process.

_process_parallel_for_loop_with_queue(*item_list: Collection[Any]*) → None

Process items in parallel using a producer/consumer queue.

Items are processed in worker threads, while a dedicated consumer thread handles the post-processing hooks.

Parameters
item_list (*Collection[Any]*) – The list of items to process.

_register_parameters() → None

Register processor parameters defined as ActiveParameter instances in the class.

This private method scans the class definition for any *processor.ActiveParameter* instances and creates corresponding *processor.PassiveParameter* instances to store the actual parameter values and metadata. It ensures that all processor parameters are properly initialised and available for configuration through the processor's configuration system.

The method checks for duplicate parameter names and raises a *ProcessorParameterError* if duplicates are detected. It also sets the internal flag *processor.Processor._parameter_registered* to True once registration is complete.

Note

This method is automatically called during processor initialisation and should not be called directly by users.

See also

processor.Processor, *processor.Processor._override_defaults()*, *processor.Processor._load_parameter_configuration()*, *processor.Processor._overrule_kws_parameters()*

Changed in version v2.0.0: Only *processor.ActiveParameter* are not registered. The use of *processor.PassiveParameter* is only meant to store the value and metadata of the active counterpart.

_remove_orphan_files() → None

Remove orphan files.

If a connection to the database is available, then the OrphanFile standard table is queried for all its entries, and all the files are then removed.

The user can turn off this behaviour by switching the *remove_orphan_files* to False.

_reset_parameters() → None

Reset processor parameters to their initial state.

This method clears all currently registered processor parameters and triggers a fresh registration process. It's useful when parameter configurations need to be reinitialized or when parameters have been modified and need to be reset.

See also

_register_parameters(), *_register_parameters()*

_reset_super_call_flag(*method: str*) → None

Reset the super-call flag for a method.

_thread_loop_context(*i_item: int*, *n_item: int*, *item: Any*) → Iterator[None]

Context manager to set thread-local loop attributes for parallel execution.

Parameters

- **i_item** (*int*) – Item index for the loop.
- **n_item** (*int*) – Total number of items in the loop.
- **item** (*Any*) – Item payload.

accept_item() → None

Does post process actions on a successfully processed item.

Within the *process()*, the user left the looping status to Continue, so it means that everything looks good and this is the right place to perform database updates or file savings.

In parallel for loops, the method can optionally accept a *LoopResult* parameter for direct access to the processed payload and looping status.

acquire_resources() → None

Acquires resources and add them to the resource stack.

The whole body of the *execute()* method is within a context structure. The idea is that if any part of the code inside should throw an exception that breaking the execution, we want to be sure that all stateful resources are properly closed.

Since the number of resources may vary, the variable number of nested *with* statements has been replaced by an ExitStack. Resources, like open files, timers, db connections, need to be added to the resource stacks in this method.

In the case a processor is being executed within a *ProcessorList*, then some resources might be shared, and for this reason they are not added to the stack. This selection can be done via the private *local_resource_acquisition*. This is normally True, meaning that the processor will handle its resources independently, but when the processor is executed from a *ProcessorList*, this flag is automatically turned to False.

If the user wants to add additional resources, he has to overload this method calling the super to preserve the original resources. If he wants to have shared resources among different processors executed from inside a processor list, he has to overload the *ProcessorList* class as well.

consumer_finish() → None

Executes once in the consumer thread after the queue has been drained.

consumer_process(*loop_result:* LoopResult | *None* = *None*) → None

Handle processed items in the consumer thread for the parallel queue loop.

By default this dispatches to *accept_item()* or *skip_item()* based on the looping status. The method can optionally accept a *LoopResult* payload.

consumer_start() → None

Executes once in the consumer thread for the parallel queue loop.

This hook mirrors *start()* but is only used with the queue-based parallel loop.

delete_parameter(*name: str*) → None

Deletes a processor parameter.

Parameters

name (*str*) – The name of the parameter to be deleted.

Raises

ProcessorParameterError – If a parameter with *name* is not registered.

dump_parameter_configuration(*option: int = 1*) → dict[str, Any]

Dumps the processor parameter values in a dictionary.

The snippet below explains the meaning of *option*.

```
# option 1
conf_dict1 = {
    'Processor': {'param1': 5, 'input_table': 'my_table'}
}

# option 2
conf_dict2 = {'param1': 5, 'input_table': 'my_table'}
```

In the case of option 1, the replica aware name (*replica_name()*) will be used as a key for the configuration dictionary.

Changed in version v2.0.0: With option 1, using *replica_name()* instead of *name* as key of the configuration dictionary.

Parameters
option (*int, Optional*) – Select the dictionary style. Defaults to 1.

Returns
A parameter configuration dictionary.

Return type
dict

execute() → None

Execute the processor tasks.

This method works as a dispatcher, reassigning the call to a more specific execution implementation depending on the *loop_type*.

finish() → None

Concludes the execution.

The user can reimplement this method if there are some conclusive tasks that must be achieved. Always include a call to super().

format_progress_message() → None

Customizes the progress message with information about the current item.

The user can overload this method in order to modify the message being displayed during the process loop with information about the current item.

The user can access the current value, its position in the looping cycle and the total number of items using *processor.Processor.item*, *processor.Processor.i_item* and *processor.Processor.n_item*.

get_filter(*model_name: str*) → *ModelFilter*

Returns a registered *ModelFilter* via the model name.

If a filter for the provided model_name does not exist, a KeyError is raised.

Parameters
model_name (*str*) – The model name for which the filter will be returned.

Returns
The registered filter

Return type
mafw.db.db_filter.ModelFilter

Raises
KeyError is a filter with the give name is not found.

get_items() → Collection[Any]

Returns the item collections for the processor loop.

This method must be overloaded for the processor to work. Generally, this is getting a list of rows from the database, or a list of files from the disk to be processed.

Returns
A collection of items for the loop

Return type
Collection[Any]

get_parameter(*name: str*) → *PassiveParameter*[Any]

Gets the processor parameter named name.

Parameters
name (*str*) – The name of the parameter.

Returns
The processor parameter

Return type
processor.PassiveParameter

Raises
ProcessorParameterError – If a parameter with *name* is not registered.

get_parameters() → dict[str, *PassiveParameter*[Any]]

Returns the full dictionary of registered parameters for this processor.

Useful when dumping the parameter specification in a configuration file, for example.

Returns
The dictionary with the registered parameters.

Return type
dict[str, *processor.PassiveParameter*[*ParameterType*]

initialise_parameters() → None

Initialises processor parameters by registering them and applying various configuration sources.

This method orchestrates the parameter initialisation process by performing the following steps in order:

1. Registers processor parameters defined as *processor.ActiveParameter* instances
2. Overrides default parameter values with any configured overrides
3. Loads parameter configuration from the processor's configuration dictionary
4. Applies keyword arguments as parameter overrides

The method ensures that all processor parameters are properly configured before the processor execution begins. It is automatically called during processor initialisation and should not typically be called directly by users.

See also

_register_parameters(), *_override_defaults()*, *_load_parameter_configuration()*, *_overrule_kws_parameters()*

Added in version v2.0.0.

on_looping_status_set(*status:* LoopingStatus) → None

Call back invoked when the looping status is set.

The user can overload this method according to the needs.

Parameters
status (LoopingStatus) – The set looping status.

on_processor_status_change(*old_status:* ProcessorStatus, *new_status:* ProcessorStatus) → None

Callback invoked when the processor status is changed.

Parameters

- **old_status** (ProcessorStatus) – The old processor status.
- **new_status** (ProcessorStatus) – The new processor status.

print_process_statistics() → None

Print the process statistics.

A utility method to display the fastest, the slowest and the average timing required to process on a single item. This is particularly useful when the looping processor is part of a ProcessorList.

process() → None

Processes the current item.

This is the core of the Processor, where the user has to define the calculations required.

In parallel for loops, the method can optionally accept a *LoopItem* parameter if the user prefers not to rely on thread-local access to *processor.Processor.item*, *processor.Processor.i_item* and *processor.Processor.n_item*.

set_parameter_value(*name: str*, *value:* ParameterType) → None

Sets the value of a processor parameter.

Parameters

- **name** (*str*) – The name of the parameter to be deleted.
- **value** (ParameterType) – The value to be assigned to the parameter.

Raises
ProcessorParameterError – If a parameter with *name* is not registered.

skip_item() → None

Does post process actions on a *NOT* successfully processed item.

Within the *process()*, the user set the looping status to Skip, so it means that something went wrong and here corrective actions can be taken if needed.

In parallel for loops, the method can optionally accept a *LoopResult* parameter for direct access to the processed payload and looping status.

start() → None

Start method.

The user can overload this method, including all steps that should be performed at the beginning of the operation.

If the user decides to overload it, it should include a call to the super method.

validate_configuration() → None

Validate the configuration provided via the processor parameters.

Method to be implemented by subclasses if a configuration validation is needed.

The method should silently check for the proper configuration, if this is not obtained, then the *InvalidConfigurationError* must be raised.

Added in version v2.0.0.

while_condition() → bool

Return the while condition

Returns
True if the while loop has to continue, false otherwise.

Return type
bool

_config: dict[str, Any]

A dictionary containing the processor configuration object.

This dictionary is populated with configuration parameter (always type 2) during the *_load_parameter_configuration()* method.

The original value of the configuration dictionary that is passed to the constructor is stored in *_orig_config*.

Changed in version v2.0.0: Now it is an empty dictionary until the *_load_parameter_configuration()* is called.

_ids = count(0)

A counter for all processor instances

_item: Any

The current item of the loop.

_looping_status: *LoopingStatus*

The looping status for the main thread.

_methods_to_be_checked_for_super: tuple[str, ...] = ('start', 'finish')

List of methods that should invoke their super implementation when overridden.

_orig_config

A copy of the original configuration dictionary.

Added in version v2.0.0.

_parameter_registered

A boolean flag to confirm successful parameter registration.

_processor_parameters: OrderedDict[str, *PassiveParameter*[Any]]

A dictionary to store all the processor parameter instances.

The name of the parameter is used as a key, while for the value an instance of the *processor.PassiveParameter* is used.

_super_call_flags: dict[str, bool]

Tracks super-call usage for methods that require it.

_thread_local

Thread-local storage for loop attributes in parallel execution.

Added in version v2.1.0.

_wall_clock_start: float | None

Timestamp when the looping execution began.

create_standard_tables

The boolean flag to proceed or skip with standard table creation and initialisation

property database: Database

Returns the database instance

Returns
A database object.

Raises
MissingDatabase – If the database connection has not been established.

description

A short description of the processor task.

filter_register: *ProcessorFilter*

The DB filter register of the Processor.

property i_item: int

The enumeration of the current item being processed.

property item: Any

The current item of the loop.

property local_resource_acquisition: bool

Checks if resources should be acquired locally.

When the processor is executed in stand-alone mode, it is responsible to acquire and release its own external resources, but when it is executed from a ProcessorList, then is a good practice to share and distribute resources among the whole processor list. In this case, resources should not be acquired locally by the single processor, but from the parent execution context.

Returns
True if resources are to be acquired locally by the processor. False, otherwise.

Return type
bool

loop_type: *LoopType*

The loop type.

The value of this parameter can also be changed by the *execution_workflow()* decorator factory.

See *LoopType* for more details.

property looping_status: *LoopingStatus*

The looping status for the current thread context.

max_workers

Maximum number of worker threads used in parallel loops.

property n_item: int | None

The total number of items to be processed or None for an undefined loop

name

The name of the processor.

new_defaults: dict[str, Any] = {}

A dictionary containing defaults value for the parameters to be overridden

Added in version v2.0.

processor_exit_status

Processor exit status

processor_status

Processor execution status

progress_message: str = 'Processor is working'

Message displayed to show the progress.

It can be customized with information about the current item in the loop by overloading the *format_progress_message()*.

queue_batch_size

Number of items processed per worker task in *ParallelForLoopWithQueue*.

queue_size

Maximum size of the queue used by *ParallelForLoopWithQueue*.

remove_orphan_files: bool

The flag to remove or protect the orphan files. Defaults to True

replica_id

The replica identifier specified in the constructor

Added in version v2.0.0.

property replica_name: str

Returns the replica aware name of the processor.

If no replica_id is specified, then return the pure name, otherwise join the two string using the '#' symbol.

Added in version v2.0.0.

Returns
The replica aware name of the processor.

Return type
str

unique_id

A unique identifier representing how many instances of Processor has been created.

property unique_name: str

Returns the unique name for the processor.

mafw.processor.meta

Metaclass for the Processor framework.

This module defines *ProcessorMeta*, the metaclass responsible for finalising every *Processor* subclass at creation time. Its duties include:

- Ensuring parameter definitions are properly inherited and registered.
- Validating any declared filter schema against the allowed model constraints.
- Installing the super-call wrappers that guard method execution at runtime.
- Invoking `__post_init__()` after each instance is constructed.

Added in version 2.3: Extracted from the monolithic `processor.py` module for improved maintainability and focused testing.

Classes

ProcessorMeta(name, bases, namespace)	Metaclass that finalizes Processor subclasses before instantiation.

class mafw.processor.meta.**ProcessorMeta**(*name: str*, *bases: tuple[type, ...]*, *namespace: dict[str, Any]*)

Bases: `type`

Metaclass that finalizes Processor subclasses before instantiation.

Set up new Processor subclasses.

This method first checks the class for well-formed parameter definitions and filter metadata, then runs the super-call wrapping machinery described below. The wrappers are installed at class-creation time so that any later instance will be guarded before it even starts executing.

mafw.processor.parameters

Parameter descriptors for the Processor framework.

This module defines the parameter descriptor system that underpins all *Processor* subclass configuration. It provides:

- *PassiveParameter* — the private storage backing each parameter.
- *ActiveParameter* — the public descriptor interface for declaring processor parameters in a class body.
- *ensure_parameter_registration()* — a decorator guaranteeing parameters are registered before access.
- *ParameterType* — a generic type variable binding the parameter value.
- *_ParameterRegistry* — the ordered mapping of parameter names to their descriptors.

The helper functions *_copy_parent_parameter_definitions()*, *_ensure_parameter_definitions()*, and *_validate_filter_schema()* are used by the metaclass to finalise parameter bookkeeping at class-creation time.

Added in version 2.3: Extracted from the monolithic `processor.py` module for improved maintainability and focused testing.

Module Attributes

ParameterType	Generic variable type for the *ActiveParameter* and *PassiveParameter*.

Functions

ensure_parameter_registration(func)	Decorator to ensure that processor parameters are registered before *func* executes.

Classes

ActiveParameter(name[, value, default, help_doc])	The public interface to the processor parameter.
PassiveParameter(name[, value, default, ...])	An helper class to store processor parameter value and metadata.

class mafw.processor.parameters.**ActiveParameter**(*name: str*, *value:* ParameterType | *None* = *None*, *default:* ParameterType | *None* = *None*, *help_doc: str* = *''*)

Bases: Generic[*ParameterType*]

The public interface to the processor parameter.

The behaviour of a *processor.Processor* can be customised by using processor parameters. The value of these parameters can be either set via a configuration file or directly when creating the class.

If the user wants to benefit from this facility, they have to add in the instance of the Processor subclass an ActiveParameter instance in this way:

```
class MyProcessor(Processor):

    # this is the input folder
    input_folder = ActiveParameter('input_folder', Path(r'C:\'), help_doc='This␣
↪is where to look for input files')

    def __init__(self, *args, **kwargs):
        super().__init(*args, **kwargs)

        # change the input folder to something else
        self.input_folder = Path(r'D:\data')

        # get the value of the parameter
        print(self.input_folder)
```

The ActiveParameter is a descriptor, it means that when you create one of them, a lot of work is done behind the scene.

In simple words, a processor parameter is made by two objects: a public interface where the user can easily access the value of the parameter and a private interface where all other information (default, documentation…) is also stored.

The user does not have to take care of all of this. When a new ActiveParameter instance is added to the class as in the code snippet above, the private interface is automatically created and will stay in the class instance until the end of the class lifetime.

To access the private interface, the user can use the *processor.Processor.get_parameter()* method using the parameter name as a key.

The user can assign to an ActiveParameter almost any name. There are just a few invalid parameter names that are used for other purposes. The list of reserved names is available *here*. Should the user inadvertently use a reserved named, a *ProcessorParameterError* is raised.

See also

The private counterpart in the *processor.PassiveParameter*.

An explanation on how processor parameters work and should be used is given in *Understanding processor parameters*

The list of *reserved names*.

Constructor parameters:

Parameters

- **name** (*str*) – The name of the parameter.
- **value** (ParameterType, *Optional*) – The initial value of the parameter. Defaults to None.
- **default** (ParameterType, *Optional*) – The default value of the parameter, to be used when `value` is not set., Defaults to None.
- **help_doc** (*str, Optional*) – An explanatory text describing the parameter.

static _matches_container(*annotation: type | None*, *default_value: Any*, *container: type*) → bool

Generic check for whether annotation/default matches a given container type.

_is_dict_annotation(*annotation: type | None*, *default_value: Any*) → bool

Check whether the annotation or default value indicates a dict type.

_is_list_annotation(*annotation: type | None*, *default_value: Any*) → bool

Check whether the annotation or default value indicates a list type.

_resolve_parameter_annotation() → type | None

Resolve the type annotation for this parameter from the owner class hints.

_schema_default_value() → Any

Return a safe copy of the default/initial value for schema reporting.

_validate_name(*proposed_name: str*) → str

Validate that the proposed parameter name is not in the list of forbidden names.

This private method checks if the provided name is allowed for use as a processor parameter. Names that are listed in *reserved_names* cannot be used as parameter names.

Parameters
proposed_name (*str*) – The name to be validated for use as a processor parameter.

Returns
The validated name if it passes the forbidden names check.

Return type
str

Raises
ProcessorParameterError – If the proposed name is in the list of forbidden names.

to_schema() → *ParameterSchema*

Returns the static schema describing this parameter.

The schema is derived solely from the descriptor metadata and does not instantiate the owning processor.

reserved_names: list[str] = ['__logic__', '__filter__', '__new_only__', '__inheritance__', '__enable__']

A list of names that cannot be used as processor parameter names.

- *__logic__*
- *__filter__*
- *__new_only__*
- *__inheritance__*
- *__enable__*

class mafw.processor.parameters.**ParameterType**

Generic variable type for the *ActiveParameter* and *PassiveParameter*.

alias of TypeVar('ParameterType')

class mafw.processor.parameters.**PassiveParameter**(*name: str*, *value:* ParameterType | *None = None*, *default:* ParameterType | *None = None*, *help_doc: str = ''*)

Bases: Generic[*ParameterType*]

An helper class to store processor parameter value and metadata.

This class is the private interface used by the *ActiveParameter* descriptor to store its value and metadata.

When a new *ActiveParameter* is added to a class, an instance of a PassiveParameter is added to the processor parameter *register*.

See also

An explanation on how processor parameters work and should be used is given in *Understanding processor parameters*

Changed in version v2.0.0: User should only use *ActiveParameter* and never manually instantiate *PassiveParameter*.

Constructor parameters:

Parameters

- **name** (*str*) – The name of the parameter. It must be a valid python identifier.
- **value** (*ParameterType, Optional*) – The set value of the parameter. If None, then the default value will be used. Defaults to None.
- **default** (*ParameterType, Optional*) – The default value for the parameter. It is used if the *value* is not provided. Defaults to None.
- **help_doc** (*str, Optional*) – A brief explanation of the parameter.

Raises

ProcessorParameterError – if both *value* and *default* are not provided or if *name* is not a valid identifier.

property is_optional: bool

Property to check if the parameter is optional.

Returns

True if the parameter is optional

Return type

bool

property is_set: bool

Property to check if the value has been set.

It is useful for optional parameter to see if the current value is the default one, or if the user set it.

property value: *ParameterType*

Gets the parameter value.

Returns

The parameter value.

Return type

ParameterType

Raises

ProcessorParameterError – if both value and default were not defined.

class mafw.processor.parameters.**_F**

Type variable for generic callable with any return value (private to avoid Sphinx collision).

alias of TypeVar('_F', bound=Callable[[...], Any])

mafw.processor.parameters.**_copy_parent_parameter_definitions**(*owner: type[*mafw.processor.Processor*]*) → _ParameterRegistry

Build an ordered dictionary of ActiveParameters inherited from base classes.

Walks the MRO (skipping the class itself) and collects all parameter descriptors declared on ancestor classes. This ensures that subclasses inherit parameters defined higher in the hierarchy.

Parameters
owner – The Processor subclass whose ancestors are inspected.

Returns
Aggregated parameter definitions from all base classes.

Return type
_ParameterRegistry

mafw.processor.parameters.**_ensure_parameter_definitions**(*owner: type[mafw.processor.Processor]*) → _ParameterRegistry

Return the class-level parameter registry, creating it if necessary.

If the class does not yet have its own `_parameter_definitions` dict (i.e. it has not been touched by the metaclass), one is created by copying definitions from the parent classes.

Parameters
owner (*type[mafw.processor.Processor]*) – The Processor subclass to inspect/initialise.

Returns
The parameter registry for *owner*.

Return type
_ParameterRegistry

mafw.processor.parameters.**_validate_filter_schema**(*owner: type[mafw.processor.Processor]*) → None

Validate the optional *FilterSchema* declared on a Processor.

Checks that:

- `_filter_schema` is an instance of `FilterSchema` (if present).
- `root_model` inherits from *MAFwBaseModel*.
- All entries in `allowed_models` are unique MAFwBaseModel subclasses.

Parameters
owner (*type[mafw.processor.Processor]*) – The Processor subclass being validated.

Raises
ProcessorParameterError – If any of the above constraints are violated.

mafw.processor.parameters.**ensure_parameter_registration**(*func: _F*) → *_F*

Decorator to ensure that processor parameters are registered before *func* executes.

This is applied to methods that access `self._processor_parameters` and need the registration step to have completed first.

Parameters
func (_F) – The method to wrap.

Returns
The wrapped method.

Return type
_F

Raises
ProcessorParameterError – If applied to something that is not a Processor instance method.

mafw.processor.parameters._ParameterRegistry

Ordered mapping of external parameter names to their *ActiveParameter* descriptors.

alias of OrderedDict[str, ActiveParameter[Any]]

mafw.processor.processor_list

ProcessorList container for orchestrating sequences of processors.

This module defines *ProcessorList*, a specialised list that holds *Processor* instances (or nested ProcessorLists) and executes them sequentially. It manages shared resources — timer, user interface, and database connection — distributing them to each processor before execution.

Scientists typically construct a *ProcessorList* in their analysis script, populate it with processor instances representing successive analysis steps, and call *execute()* to run the full pipeline.

Added in version 2.3: Extracted from the monolithic `processor.py` module for improved maintainability and focused testing.

Classes

ProcessorList(*args[, name, description, ...])	A list like collection of processors.

class mafw.processor.processor_list.**ProcessorList**(**args: Processor | ProcessorList*, *name: str | None = None*, *description: str | None = None*, *timer: Timer | None = None*, *timer_params: dict[str, Any] | None = None*, *user_interface: UserInterfaceBase | None = None*, *database: Database | None = None*, *database_conf: dict[str, Any] | None = None*, *create_standard_tables: bool = True*)

Bases: list[Processor | ProcessorList]

A list like collection of processors.

ProcessorList is a subclass of list containing only Processor subclasses or other ProcessorList.

An attempt to add an element that is not a Processor or a ProcessorList will raise a TypeError.

Along with an iterable of processors, a new processor list can be built using the following parameters.

Constructor parameters:

Parameters

- **name** (*str, Optional*) – The name of the processor list. Defaults to ProcessorList.
- **description** (*str, Optional*) – An optional short description. Default to ProcessorList.
- **timer** (*Timer, Optional*) – The timer object. If None is provided, a new one will be created. Defaults to None.
- **timer_params** (*dict, Optional*) – A dictionary of parameter to build the timer object. Defaults to None.
- **user_interface** (*UserInterfaceBase, Optional*) – A user interface. Defaults to None
- **database** (*Database, Optional*) – A database instance. Defaults to None.
- **database_conf** (*dict, Optional*) – Configuration for the database. Default to None.
- **create_standard_tables** (*bool, Optional*) – Whether or not to create the standard tables. Defaults to True.

static validate_item(*item:* Processor | ProcessorList) → *Processor* | *ProcessorList*

Validates the item being added.

static validate_items(*items: tuple[*Processor | ProcessorList*, ...] = ()*) → tuple[*Processor* | *ProcessorList*, ...]

Validates a tuple of items being added.

acquire_resources() → None

Acquires external resources.

The resource acquisition strategy mirrors *processor.Processor.acquire_resources()*: if a resource (timer, database) is already provided, it is reused; otherwise a new one is created and registered with the exit stack for automatic cleanup.

append(*_ProcessorList__object:* Processor | ProcessorList) → None

Appends a new processor at the end of the list.

distribute_resources(*processor:* Processor | *Self*) → None

Distributes the external resources to the items in the list.

execute() → *ProcessorExitStatus*

Execute the list of processors.

Similarly to the *processor.Processor*, ProcessorList can be executed. In simple words, the execute method of each processor in the list is called exactly in the same sequence as they were added.

extend(*_ProcessorList__iterable: Iterable[*Processor | ProcessorList*]*) → None

Extends the processor list with a list of processors.

insert(*_ProcessorList__index: SupportsIndex*, *_ProcessorList__object:* Processor | ProcessorList) → None

Adds a new processor at the specified index.

create_standard_tables

The boolean flag to proceed or skip with standard table creation and initialisation

property database: Database

Returns the database instance

Returns
A database instance

Raises
MissingDatabase – if a database connection is missing.

property name: str

The name of the processor list

Returns
The name of the processor list

Return type
str

nested_list

Boolean flag to identify that this list is actually inside another list.

Similarly to the local resource flag for the *base.Processor*, this flag prevent the user interface to be added to the resource stack.

property processor_exit_status: *ProcessorExitStatus*

The processor exit status.

It refers to the whole processor list execution.

mafw.processor.utils

Standalone utility helpers for the Processor package.

This module provides validation helpers that are used across the processor sub-package but do not belong to any specific class. Currently it contains *validate_database_conf()*, which checks that a steering-file database section carries the minimum required fields before a database connection is attempted.

Added in version 2.3: Extracted from the monolithic `processor.py` module for improved maintainability and focused testing.

Functions

validate_database_conf([database_conf])	Validate the database configuration dictionary.

mafw.processor.utils.**validate_database_conf**(*database_conf: dict[str, Any] | None = None*) → dict[str, Any] | None

Validate the database configuration dictionary.

This function is called during processor initialisation to ensure that the supplied database configuration contains the minimum required fields (currently just `URL`). If the configuration is a full steering file, the `DBConfiguration` section is extracted automatically.

Parameters
database_conf (*dict, Optional*) – The input database configuration. Defaults to None.

Returns
Either the validated database configuration or None if it is invalid.

Return type
dict, None

14.1.14 mafw.processor_library

Library of generic processors.

Subpackage containing a library of general purpose processors that the user can either use directly or overload them to implement their analytical tasks.

Modules

abstract_plotter	Module implements the abstract base interface to a processor to generate plots.
db_init	Database initialisation processor module.
importer	Provides a basic element importer.
sns_plotter	Module implements a Seaborn plotter processor with a mixin structure to generate seaborn plots.

mafw.processor_library.abstract_plotter

Module implements the abstract base interface to a processor to generate plots.

This abstract interface is needed because MAFw does not force the user to select a specific plot and data manipulation library.

The basic idea is to have a *basic processor class* featuring a modified *process()* method where a skeleton of the standard operations required to generate a graphical representation of a dataset is provided.

The user has the possibility to compose the *GenericPlotter* by mixing it with one *DataRetriever* and a *FigurePlotter*.

For a specific implementation based on seaborn, please refer to *sns_plotter*.

Classes

DataRetriever(*args, **kwargs)	Base mixin class to retrieve a data frame from an external source
FigurePlotter()	
GenericPlotter(*args, **kwargs)	The Generic Plotter processor.
PlotterMeta(name, bases, namespace, /, **kwargs)	Metaclass for the plotter mixed classes

class mafw.processor_library.abstract_plotter.**DataRetriever**(**args: Any*, ***kwargs: Any*)

Bases: ABC

Base mixin class to retrieve a data frame from an external source

The dataframe instance. It will be filled for the main class

abstractmethod get_data_frame() → None

The mixin implementation of the shared method with the base class

abstractmethod patch_data_frame() → None

The mixin implementation of the shared method with the base class

class mafw.processor_library.abstract_plotter.**FigurePlotter**

Bases: ABC

class mafw.processor_library.abstract_plotter.**GenericPlotter**(**args: Any*, ***kwargs: Any*)

Bases: *Processor*

The Generic Plotter processor.

This is a subclass of a Processor with advanced functionality to fetch data in the form of a dataframe and to produce plots. When mentioning dataframe in the context of the generic plotter, we do not have in mind any specific dataframe implementation.

The GenericPlotter is actually a kind of abstract class: since MAFw is not forcing you to use any specific plotting and data manipulation library, you need to subclass the GenericPlotter in your code, be sure that the required dependencies are available for import and use it as a normal processor.

If you are ok with using seaborn (with matplotlib as a graphical backend and pandas for data storage and manipulation), then be sure to install mafw with the optional feature *seaborn* (`pip install mafw[seaborn]`) and have a look at the *sns_plotter* for an already prepared implementation of a Plotter.

The key difference with respect to a normal processor is its *process()* method that has been already implemented as follows:

```
def process(self) -> None:
    """
    Process method overload.

    In the case of a plotter subclass, the process method is already implemented␣
↪and the user should not overload
    it. On the contrary, the user must overload the other implementation methods␣
↪described in the general
    :class:`class description <.SNSPlotter>`.
    """
    if self.filter_register.new_only:
        if self.is_output_existing():
            return

    self.in_loop_customization()
    self.get_data_frame()
```

(continues on next page)

(continued from previous page)

```
    self.patch_data_frame()
    self.slice_data_frame()
    self.group_and_aggregate_data_frame()
    if not self.is_data_frame_empty():
        self.plot()
        self.customize_plot()
        self.save()
        self.update_db()
```

This actually means that when you are subclassing a GenericPlotter you do not have to implement the process method as you would do for a normal Processor, but you will have to implement the following methods:

- *in_loop_customization()*.

  The processor execution workflow (LoopType) can be any of the available, so actually the process method might be invoked only once, or multiple times inside a loop structure (for or while). If the execution is cyclic, then you may want to have the possibility to do some customisation for each iteration, for example, changing the plot title, or modifying the data selection, or the filename where the plots will be saved.

  You can use this method also in case of a single loop processor, in this case you will not have access to the loop parameters.

- *get_data_frame()*.

  This method has the task to get the data to be plotted. Since it is an almost abstract class, you need to

- *patch_data_frame()*.

  A convenient method to apply data frame manipulation to the data just retrieved. A typical use case is for conversion of unit of measurement. Imagine you saved the data in the S.I. units, but for the visualization you prefer to use practical units, so you can subclass this method to add a new column containing the same converted values of the original one.

- *slice_data_frame()*.

  Slicing a dataframe is similar as applying a where clause in a SQL query. Implement this method to select which row should be used in the generation of your plot.

- *group_and_aggregate_data_frame()*.

  In this method, you can manipulate your data frame to perform row grouping and aggregation.

- *is_data_frame_empty()*.

  A simple method to test if the dataframe contains any data to be plotted. In fact, after the slicing, grouping and aggregation operations, it is possible that the dataframe is now left without any row. In this case, it makes no sense to waste time in plotting an empty graph.

- *plot()*.

  This method is where the actual plotting occurs.

- *customize_plot()*.

  This method can be optionally used to customize the appearance of the facet grid produced by the *plot()* method. It is particularly useful when the user is mixing this class with one of the *FigurePlotter* mixin, thus not having direct access to the plot method.

- *save()*.

  This method is where the produced plot is saved in a file. Remember to append the output file name to the *list of produced outputs* so that the *_update_plotter_db()* method will automatically store this file in the database during the *finish()* execution.

- *update_db()*.

If the user wants to update a specific table in the database, they can use this method.

It is worth reminding that all plotters are saving all generated files in the standard table PlotterOutput. This is automatically done by the *_update_plotter_db()* method that is called in the *finish()* method.

Processor parameters

- **force_replot**: Whether to force re-plotting even if the output file already exists (default: False)
- **output_folder**: The path where the output file will be saved (default: PosixPath('/tmp/mafw-docs-kpuvl5nc/v2.3.0_pdf'))

Constructor parameters

Parameters

- **name** (*str, Optional*) – The name of the processor. If None is provided, the class name is used instead. Defaults to None.
- **description** (*str, Optional*) – A short description of the processor task. Defaults to the processor name.
- **config** (*dict, Optional*) – A configuration dictionary for this processor. Defaults to None.
- **looper** (*LoopType, Optional*) – Enumerator to define the looping type. Defaults to LoopType.ForLoop
- **user_interface** (*UserInterfaceBase, Optional*) – A user interface instance to be used by the processor to interact with the user.
- **timer** (*Timer, Optional*) – A timer object to measure process duration.
- **timer_params** (*dict, Optional*) – Parameters for the timer object.
- **database** (*Database, Optional*) – A database instance. Defaults to None.
- **database_conf** (*dict, Optional*) – Configuration for the database. Default to None.
- **remove_orphan_files** (*bool, Optional*) – Boolean flag to remove files on disc without a reference to the database. See *Standard tables* and *_remove_orphan_files()*. Defaults to True
- **replica_id** (*str, Optional*) – The replica identifier for the current processor.
- **create_standard_tables** (*bool, Optional*) – Boolean flag to create std tables on disk. Defaults to True When a nested steering configuration is loaded, this value can be overridden by the global create_standard_tables entry. Flat processor configurations keep the constructor value.
- **max_workers** (*int, Optional*) – Number of worker threads for parallel loops.
- **queue_size** (*int, Optional*) – Maximum size of the internal queue for the queue-based parallel loop.
- **queue_batch_size** (*int, Optional*) – Number of items processed per worker task in the queue-based parallel loop.
- **kwargs** – Keyword arguments that can be used to set processor parameters.

_update_plotter_db() → None

Updates the Plotter DB.

A plotter subclass primarily generates plots as output in most cases, which means that no additional information needs to be stored in the database. This is sufficient to prevent unnecessary execution of the processor when it is not required.

This method is actually protected against execution without a valid database instance.

Changed in version v2.0.0: Using the *processor.Processor.replica_name* instead of the *processor.Processor.name* as plotter_name in the *PlotterOutput* Model.

customize_plot() → None

The customize plot method.

The user can overload this method to customize the output produced by the *plot()* method, like, for example, adding meaningful axis titles, changing format, and so on.

As usual, it is possible to use the *item*, *i_item* and *n_item* to access the loop parameters.

finish() → None

Concludes the execution.

The user can reimplement this method if there are some conclusive tasks that must be achieved. Always include a call to super().

format_progress_message() → None

Customizes the progress message with information about the current item.

The user can overload this method in order to modify the message being displayed during the process loop with information about the current item.

The user can access the current value, its position in the looping cycle and the total number of items using *processor.Processor.item*, *processor.Processor.i_item* and *processor.Processor.n_item*.

get_data_frame() → None

Get the data frame with the data to be plotted.

This method can be either implemented in the SNSPlotter subclass or via a *DataRetriever* mixin class.

in_loop_customization() → None

Customize the parameters for the output or input data for each execution iteration.

is_data_frame_empty() → bool

Check if the data frame is empty

is_output_existing() → bool

Check for plotter output existence.

Generally, plotter subclasses do not have a real output that can be saved to a database. This class is meant to generate one or more graphical output files.

One of the biggest advantages of having the output of a processor stored in the database is the ability to conditionally execute the processor if, and only if, the output is missing or changed.

In order to allow also plotter processor to benefit from this feature, a *dedicated table* is available among the *standard tables*.

If a connection to the database is provided, then this method is invoked at the beginning of the *process()* and a select query over the *PlotterOutput* model is executed filtering by processor name. All files in the filename lists are checked for existence and also the checksum is verified.

Especially during debugging phase of the processor, it is often needed to generate the plot several times, for this reason the user can switch the *force_replot* parameter to True in the steering file and the output file will be generated even if it is already existing.

This method will return True, if the output of the processor is already existing and valid, False, otherwise.

Changed in version v2.0.0: Using *processor.Processor.replica_name* instead of *processor.Processor.name* for storage in the *PlotterOutput*

Returns

True if the processor output exists and it is valid.

Return type
bool

patch_data_frame() → None

Modify the data frame

This method can be used to perform operation on the data frame, like adding new columns. It can be either implemented in the plotter processor subclasses or via a mixin class.

plot() → None

The plot method.

This is where the user has to implement the real plot generation

process() → None

Process method overload.

In the case of a plotter subclass, the process method is already implemented and the user should not overload it. On the contrary, the user must overload the other implementation methods described in the general *class description*.

save() → None

The save method.

This is where the user has to implement the saving of the plot on disc.

update_db() → None

The update database method.

This is where the user has to implement the optional update of the database.

force_replot

Flag to force the regeneration of the output file even if it is already existing.

class mafw.processor_library.abstract_plotter.**PlotterMeta**(*name*, *bases*, *namespace*, /, ***kwargs*)

Bases: _ProtocolMeta, *ProcessorMeta*

Metaclass for the plotter mixed classes

mafw.processor_library.db_init

Database initialisation processor module.

This module contains the following processors:

TableCreator
: processor which handles the creation of database tables based on registered models. It provides functionality to create tables automatically while respecting existing tables and offering options for forced recreation.

TriggerRefresher
: processor to safely update the trigger definitions. It removes all existing triggers and regenerates them according to the new definition. Particularly useful when debugging triggers, it can also be left at the beginning of all analysis pipelines since it does not cause any loss of data.

SQLScriptRunner
: processor to execute SQL scripts from files against the database. It reads SQL files, removes block comments, splits the content into individual statements, and executes them within a transaction.

Added in version v2.0.0.

Classes

SQLScriptRunner(*args, **kwargs)	Processor to execute SQL scripts from files against the database.
TableCreator(*args, **kwargs)	Processor to create all tables in the database.
TriggerRefresher(*args, **kwargs)	Processor to recreate all triggers.

class mafw.processor_library.db_init.**SQLScriptRunner**(**args: Any*, ***kwargs: Any*)

Bases: *Processor*

Processor to execute SQL scripts from files against the database.

This processor reads SQL files, removes multi-line block comments, splits the content into individual statements, and executes them within a transaction. It is designed to handle SQL script execution in a safe manner by wrapping all statements in a single atomic transaction.

The processor accepts a list of SQL files through the *sql_files* parameter. Each file is validated to ensure it exists and is a regular file before processing. Block comments (/* ... */) are removed from the SQL content before statement parsing.

Added in version v2.0.0.

Processor parameters

- **sql_files**: A list of SQL files to be processed (default: [])

Constructor parameters

Parameters

- **name** (*str, Optional*) – The name of the processor. If None is provided, the class name is used instead. Defaults to None.
- **description** (*str, Optional*) – A short description of the processor task. Defaults to the processor name.
- **config** (*dict, Optional*) – A configuration dictionary for this processor. Defaults to None.
- **looper** (*LoopType, Optional*) – Enumerator to define the looping type. Defaults to LoopType.ForLoop
- **user_interface** (*UserInterfaceBase, Optional*) – A user interface instance to be used by the processor to interact with the user.
- **timer** (*Timer, Optional*) – A timer object to measure process duration.
- **timer_params** (*dict, Optional*) – Parameters for the timer object.
- **database** (*Database, Optional*) – A database instance. Defaults to None.
- **database_conf** (*dict, Optional*) – Configuration for the database. Default to None.
- **remove_orphan_files** (*bool, Optional*) – Boolean flag to remove files on disc without a reference to the database. See *Standard tables* and *_remove_orphan_files()*. Defaults to True
- **replica_id** (*str, Optional*) – The replica identifier for the current processor.
- **create_standard_tables** (*bool, Optional*) – Boolean flag to create std tables on disk. Defaults to True When a nested steering configuration is loaded, this value can be overridden by the global create_standard_tables entry. Flat processor configurations keep the constructor value.
- **max_workers** (*int, Optional*) – Number of worker threads for parallel loops.
- **queue_size** (*int, Optional*) – Maximum size of the internal queue for the queue-based parallel loop.

- **queue_batch_size** (*int, Optional*) – Number of items processed per worker task in the queue-based parallel loop.
- **kwargs** – Keyword arguments that can be used to set processor parameters.

format_progress_message() → None

Customizes the progress message with information about the current item.

The user can overload this method in order to modify the message being displayed during the process loop with information about the current item.

The user can access the current value, its position in the looping cycle and the total number of items using *processor.Processor.item*, *processor.Processor.i_item* and *processor.Processor.n_item*.

get_items() → Collection[Any]

Get the collection of SQL files to be processed.

Returns
A collection of SQL file paths to be processed

Return type
Collection[Any]

process() → None

Process a single SQL file by reading, parsing, and executing its statements.

Reads the SQL file content, removes multi-line block comments, splits the content into individual SQL statements, and executes them within a transaction.

If no statements are found in the file, a warning is logged. If an error occurs during execution, the transaction is rolled back and the exception is re-raised.

Raises
Exception – If an error occurs during SQL statement execution.

start() → None

Start method.

The user can overload this method, including all steps that should be performed at the beginning of the operation.

If the user decides to overload it, it should include a call to the super method.

validate_configuration() → None

Validate the configuration of SQL script runner.

Ensures that all specified SQL files exist and are regular files.

Raises
InvalidConfigurationError – if any of the specified files does not exist or is not a regular file.

sql_files

List of SQL files to be processed

class mafw.processor_library.db_init.**TableCreator**(**args: Any*, ***kwargs: Any*)

Bases: *Processor*

Processor to create all tables in the database.

This processor can be included in all pipelines in order to create all tables in the database. Its functionality is based on the fact that all *MAFwBaseModel* subclasses are automatically included in a global register (*mafw_model_register*).

This processor will perform the following:

1. Get a list of all tables already existing in the database.

2. Prune from the lists of models the ones for which already exist in the database.
3. Create the remaining tables.

This overall behaviour can be modified via the following parameters:

- *force_recreate* (bool, default = False): Use with extreme care. When set to True, all tables in the database and in the model register will be first dropped and then recreated. It is almost equivalent to a re-initialization of the whole DB with all the data being lost.
- *soft_recreate* (bool, default = True): When set to true, all tables whose model is in the mafw model register will be recreated with the safe flag. It means that there won't be any table drop. If a table is already existing, nothing will happen. If a new trigger is added to the table this will be created. When set to False, only tables whose model is in the register and that are not existing will be created.
- *apply_only_to_prefix* (list[str], default = []): This parameter allows to create only the tables that do not already exist and whose name start with one of the provided prefixes.

Added in version v2.0.0.

Processor parameters

- **apply_only_to_prefix**: Create only tables whose name start with the provided prefixes. (default: [])
- **force_recreate**: First drop and then create the tables. LOSS OF ALL DATA!!! (default: False)
- **soft_recreate**: Safe recreate tables without dropping. No data loss (default: True)

Constructor parameters

Parameters

- **name** (*str, Optional*) – The name of the processor. If None is provided, the class name is used instead. Defaults to None.
- **description** (*str, Optional*) – A short description of the processor task. Defaults to the processor name.
- **config** (*dict, Optional*) – A configuration dictionary for this processor. Defaults to None.
- **looper** (LoopType, *Optional*) – Enumerator to define the looping type. Defaults to LoopType.ForLoop
- **user_interface** (UserInterfaceBase, *Optional*) – A user interface instance to be used by the processor to interact with the user.
- **timer** (Timer, *Optional*) – A timer object to measure process duration.
- **timer_params** (*dict, Optional*) – Parameters for the timer object.
- **database** (*Database, Optional*) – A database instance. Defaults to None.
- **database_conf** (*dict, Optional*) – Configuration for the database. Default to None.
- **remove_orphan_files** (*bool, Optional*) – Boolean flag to remove files on disc without a reference to the database. See *Standard tables* and *_remove_orphan_files()*. Defaults to True
- **replica_id** (*str, Optional*) – The replica identifier for the current processor.
- **create_standard_tables** (*bool, Optional*) – Boolean flag to create std tables on disk. Defaults to True When a nested steering configuration is loaded, this value can be overridden by the global create_standard_tables entry. Flat processor configurations keep the constructor value.
- **max_workers** (*int, Optional*) – Number of worker threads for parallel loops.
- **queue_size** (*int, Optional*) – Maximum size of the internal queue for the queue-based parallel loop.

- **queue_batch_size** (*int, Optional*) – Number of items processed per worker task in the queue-based parallel loop.
- **kwargs** – Keyword arguments that can be used to set processor parameters.

process() → None

Execute the table creation process.

This method performs the following steps:

1. Identify all models that have automatic creation enabled.
2. Filter models based on the apply_only_to_prefix parameter if specified.
3. Handle forced recreation if requested, including user confirmation.
4. Handle soft recreation if requested, letting all tables with a known model be recreated.
5. Create the required tables.
6. Initialise standard tables after recreation if needed.

If user cancel the creation, the processor exit status is set to *ProcessorExitStatus.Aborted* so that the whole processor list is blocked.

start() → None

Start method.

The user can overload this method, including all steps that should be performed at the beginning of the operation.

If the user decides to overload it, it should include a call to the super method.

validate_configuration() → None

Configuration validation

force_recreate and *soft_recreate* cannot be both valid.

Raises
InvalidConfigurationError – if both recreate types are True.

apply_only_to_prefix

Apply only to tables starting with prefix (list[str], default = []).

This parameter allows to create only the tables that do not already exist and whose name start with one of the provided prefixes.

existing_table_names: list[str]

The list of all existing tables in the database.

force_recreate

Force recreate (bool, default = False).

Use with extreme care. When set to True, all tables in the database and in the model register will be first dropped and then recreated. It is almost equivalent to a re-initialization of the whole DB with all the data being lost.

soft_recreate

Soft recreate (bool default = True).

When set to true, all tables whose model is in the mafw model register will be recreated with the safe flag. It means that there won't be any table drop. If a table is already existing, nothing will happen. If a new trigger is added to the table, this will be created. When set to False, only tables whose model is in the register and that are not existing will be created.

class mafw.processor_library.db_init.**TriggerRefresher**(**args: Any*, ***kwargs: Any*)

Bases: *Processor*

Processor to recreate all triggers.

Triggers are database objects, and even though they could be created, dropped and modified at any moment, within the MAFw execution cycle they are normally created along with the table they are targeting.

When the table is created, also all its triggers are created, but unless differently specified, with the safe flag on, that means that they are created if they do not exist.

This might be particularly annoying when modifying an existing trigger, because you need to manually drop the trigger to let the table creation mechanism to create the newer version.

The goal of this processor is to drop all existing triggers and then recreate the corresponding tables so to have an updated version of the triggers.

The processor is relying on the fact that all subclasses of *MAFwBaseModel* are automatically inserted in the *mafw_model_register* so that the model class can be retrieved from the table name.

Before removing any trigger, the processor will build a list with all the affected tables and check if all of them are in the *mafw_model_register*, if so, it will proceed without asking any further confirmation. Otherwise, if some affected tables are not in the register, then it will ask the user to decide what to do:

- Remove only the triggers whose tables are in the register and thus recreated afterward.
- Remove all triggers, in this case, some of them will not be recreated.
- Abort the processor.

Trigger manipulations (drop and creation) are not directly implemented in peewee and are an extension provided by MAFw. In order to be compatible with the three main databases (sqlite, mysql and postgresql), the SQL generation is obtained via the *TriggerDialect* interface.

See also

The *Trigger* class and also the *trigger chapter* for a deeper explanation on triggers.

The *ModelRegister* class, the *mafw_model_register* and the *related chapter* on the automatic registration mechanism.

The *TriggerDialect* and its subclasses, for a database independent way to generate SQL statement related to triggers.

Added in version v2.0.0.

Constructor parameters

Parameters

- **name** (*str, Optional*) – The name of the processor. If None is provided, the class name is used instead. Defaults to None.
- **description** (*str, Optional*) – A short description of the processor task. Defaults to the processor name.
- **config** (*dict, Optional*) – A configuration dictionary for this processor. Defaults to None.
- **looper** (LoopType, *Optional*) – Enumerator to define the looping type. Defaults to LoopType.ForLoop
- **user_interface** (UserInterfaceBase, *Optional*) – A user interface instance to be used by the processor to interact with the user.
- **timer** (Timer, *Optional*) – A timer object to measure process duration.
- **timer_params** (*dict, Optional*) – Parameters for the timer object.

- **database** (*Database, Optional*) – A database instance. Defaults to None.
- **database_conf** (*dict, Optional*) – Configuration for the database. Default to None.
- **remove_orphan_files** (*bool, Optional*) – Boolean flag to remove files on disc without a reference to the database. See *Standard tables* and *_remove_orphan_files()*. Defaults to True
- **replica_id** (*str, Optional*) – The replica identifier for the current processor.
- **create_standard_tables** (*bool, Optional*) – Boolean flag to create std tables on disk. Defaults to True When a nested steering configuration is loaded, this value can be overridden by the global create_standard_tables entry. Flat processor configurations keep the constructor value.
- **max_workers** (*int, Optional*) – Number of worker threads for parallel loops.
- **queue_size** (*int, Optional*) – Maximum size of the internal queue for the queue-based parallel loop.
- **queue_batch_size** (*int, Optional*) – Number of items processed per worker task in the queue-based parallel loop.
- **kwargs** – Keyword arguments that can be used to set processor parameters.

finish() → None

Recreate the tables from which triggers were dropped.

This is only done if the user did not abort the process.

format_progress_message() → None

Customizes the progress message with information about the current item.

The user can overload this method in order to modify the message being displayed during the process loop with information about the current item.

The user can access the current value, its position in the looping cycle and the total number of items using *processor.Processor.item*, *processor.Processor.i_item* and *processor.Processor.n_item*.

get_dialect() → *TriggerDialect*

Get the valid SQL dialect based on the type of Database

Returns
The SQL trigger dialect

Type
TriggerDialect

Raises
UnsupportedDatabaseError if there is no dialect for the current DB.

get_items() → Collection[Any]

Retrieves a list of database triggers and interacts with the user to determine which ones to process.

This method fetches all currently defined database triggers. If any tables associated with these triggers are not known (i.e., not registered in *mafw_model_register*), it enters an interactive mode to prompt the user for a course of action:

1. **Remove All Triggers (A):** Processes all triggers for subsequent removal, but only marks 'rebuildable' tables for rebuilding.
2. **Remove Only Rebuildable Triggers (O):** Processes only triggers associated with 'rebuildable' tables.
3. **Quit (Q):** Aborts the entire process.

If no unknown tables are found, or the user chooses to process rebuildable tables, the list of triggers and the set of tables to be rebuilt are prepared for the next stage.

Returns
A collection of database triggers to be processed, in the for tuple trigger_name, table_name

Return type
list[tuple[str, str]]

process() → None

Delete the current trigger from its table

start() → None

Start method.

The user can overload this method, including all steps that should be performed at the beginning of the operation.

If the user decides to overload it, it should include a call to the super method.

mafw.processor_library.db_init.**_standard_table_models**() → set[type[*StandardTable*]]

Return the registered standard-table models when available.

mafw.processor_library.importer

Provides a basic element importer.

The first step in the setting up of the analytical framework of a data analysis procedure is to add new elements to the input set. These elements can encompass a wide range of data, including results from experiments or simulations, as well as information gathered through from webscraping or other data sources.

Independently of where the data are coming from, one common task is to add those data to your collection inside the DB, so that the following analytical steps know where the data are and what they are.

This module provides a generic processor that the user can subclass and customize to their needs to import input files. Thanks to a smart filename parsing, other information can be extracted from the filename itself and used to populate additional columns in the dedicated database table.

Classes

FilenameElement(name, regex, value_type, ...)	Helper class for the definition of filename element.
FilenameParser(configuration_file[, filename])	Helper class to interpret all elements in a filename.
Importer(*args, **kwargs)	Importer is the base class for importing elements in the Database structure.

class mafw.processor_library.importer.**FilenameElement**(*name: str, regex: str | ~re.Pattern[str], value_type: type = <class 'str'>, default_value: str | int | float | None = None*)

Bases: object

Helper class for the definition of filename element.

While importing an element to the DB, several parameters can be retrieved directly from the filename. The role of this helper class is to provide an easy way to define patterns in the filename representing a specific piece of information that has to be transferred to the DB.

The element is characterized by a name, a regular expression, the expected python type for the parsed value and an optional default value. The regular expression should contain a named group in the form ?P<name> where name is matching the FilenameElement name.

To make a filename element optional, it is enough to provide a default value different from None. In this case, if the parsing is failing, then the default value will be returned.

Constructor parameters:

Parameters

- **name** (*str*) – The name of the filename element
- **regex** (*str | re.Pattern[str]*) – The regular expression associated to this filename element. It must contain a named group in the form ?P<name>.
- **value_type** (*type, Optional*) – The type the output value should be converted into. It defaults to str.
- **default_value** (*Any, Optional*) – The default value to assign to the filename element if the pattern is not found in the filename. It defaults to None

classmethod _get_value_type(*type_as_string: str*) → type

Returns the value type.

This method is used by the class method constructor to check if the user provided type in the form of a string is a valid one.

If so, then the corresponding python type is returned, otherwise a ValueError exception is raised.

Parameters
type_as_string (*str*) – The type of the value as a string.

Returns
The corresponding python type.

Return type
type

Raises
ValueError – if type_as_string is not any of the acceptable type for the value.

classmethod from_dict(*name: str*, *info_dict: dict[str, str | int | float]*) → *FilenameElement*

Generates a FilenameElement starting from external information stored in a dictionary.

***info_dict* should contain the following three keys:**

- regexp: the Regular expression for the element search.
- type: a string with the python type name (int, float, str) for the element conversion.
- default (*optional*): a default value.

Parameters

- **name** (*str*) – The name of the element.
- **info_dict** (*dict*) – The dictionary with the required parameters for the class constructor.

Returns
An instance of FilenameElement.

Return type
FilenameElement

_validate_default_type() → None

Checks that the default has a type matching the value type. The check is actually performed if and only if a default value is provided. If None, then the validation is skipped.

Raises
TypeError – if the default value type does not match the declared value type.

_validate_regexp() → None

Checks if the regular expression contains a named group named after the element itself.

Raises
ValueError – if the regular expression is not valid.

reset() → None

Resets the value to the default value.

Remember: that the default value is None for compulsory elements.

search(*string: str | Path*) → None

Searches the string for the regular expression.

If the pattern is found in the string, then the matched value is transferred to the FilenameElement value.

> **Note**
>
> This method is not returning the match value. It is only searching the input string for the registered pattern. If the pattern is found, then the user can retrieve the matched value by invoking the *value()* method. If the pattern is not found, the *value()* will return either None, for a compulsory element, or the default value for an optional one.

Parameters
string (*str | Path*) – The string to be parsed. In most of the case, this is a filename, that is why the method is accepting also a Path type.

property is_found: bool

Returns if the file element is found

property is_optional: bool

Returns if the element is optional

property name: str

Returns the class name

property pattern: str | bytes

Returns the regular expression pattern

type_lut: dict[str, type[str] | type[int] | type[float]] = {'float': <class 'float'>, 'int': <class 'int'>, 'str': <class 'str'>}

A lookup table for converting type definition as string into python types

property value: str | int | float | None

Returns the class value

class mafw.processor_library.importer.**FilenameParser**(*configuration_file: str | Path*, *filename: str | Path | None = None*)

Bases: object

Helper class to interpret all elements in a filename.

Inside a filename, there might be many elements containing information about the item that must be stored in the DB. This class will parse the filename, and after a successful identification of them all, it will make them available for the importer class to fill in the fields in the database.

The *FilenameParser* needs to be configured to be able to recognise each element in the filename. Such configuration is saved in a *toml* file. An example of such a configuration is provided here.

Each element must start with its name and a valid regular expression and a python type (in string). If an element is optional, then a default value must be provided as well.

After the configuration, the filename can be interpreted invoking the *interpret()* method. This will perform the actual parsing of the filename. If an error occurs during the parsing process, meaning that a compulsory element is not found, then the *ParsingError* exception will be raised. So remember to protect the interpretation with a try/except block.

The value of each file element is available upon request. The user has simply to invoke the *get_element_value()* providing the element name.

Constructor parameters:

Parameters

- **filename** (*str | Path*) – The filename to be interpreted.
- **configuration_file** (*str | Path*) – The configuration file for the interpreter.

Raises

ParserConfigurationError – If the configuration file is invalid.

_parser_configuration() → None

Loads the parser configuration, generates the required FilenameElement and adds them element dictionary.

The configuration file is stored in a TOML file.

This private method is automatically invoked by the class constructor.

Raises

ParserConfigurationError – if the provided configuration file is invalid.

get_element(*element_name: str*) → *FilenameElement* | None

Gets the FilenameElement named element_name

get_element_value(*element_name: str*) → str | int | float | None

Gets the value of the FilenameElement named element_name.

It is equivalent to call `self.get_element('element_name').value`

interpret(*filename: str | Path | None = None*) → None

Performs the interpretation of the filename.

The filename can be provided either as constructor argument or here as an argument. If both, then the local one will have the precedence.

Raises

- *ParsingError* – if a compulsory element is not found in the filename
- *MissingAttribute* – if no filename has been specified.

reset() → None

Resets all filename elements

_configuration_file

The configuration file for the interpreter.

_element_dict: dict[str, *FilenameElement*]

A dictionary with all the FilenameElement

_filename

The filename for this interpreter. If None, it should be specified before interpretation.

property elements: dict[str, *FilenameElement*]

Returns the filename element dictionary

class mafw.processor_library.importer.**Importer**(**args: Any*, ***kwargs: Any*)

Bases: *Processor*

Importer is the base class for importing elements in the Database structure.

It provides an easy skeleton to be subclassed by a more specific importer related to a certain project.

It can be customised with three processor parameters:

- The `parser_configuration`: the path to the configuration file for the *FilenameParser*.
- The `input_folder`: the path where the input files to be imported are.
- The `recursive` flag: to specify if all subfolders should be also scanned.

For a concrete implementation, have a look at the *ImporterExample* from the example library.

Processor parameters

- **input_folder**: The input folder from where the images have to be imported. (default: '/tmp/mafw-docs-kpuvl5nc/v2.3.0_pdf')
- **parser_configuration**: The path to the TOML file with the filename parser configuration (default: 'parser_configuration.toml')
- **recursive**: Extend the search to sub-folder (default: True)

Constructor parameters

Parameters

- **name** (*str, Optional*) – The name of the processor. If None is provided, the class name is used instead. Defaults to None.
- **description** (*str, Optional*) – A short description of the processor task. Defaults to the processor name.
- **config** (*dict, Optional*) – A configuration dictionary for this processor. Defaults to None.
- **looper** (LoopType, *Optional*) – Enumerator to define the looping type. Defaults to LoopType.ForLoop
- **user_interface** (UserInterfaceBase, *Optional*) – A user interface instance to be used by the processor to interact with the user.
- **timer** (Timer, *Optional*) – A timer object to measure process duration.
- **timer_params** (*dict, Optional*) – Parameters for the timer object.
- **database** (*Database, Optional*) – A database instance. Defaults to None.
- **database_conf** (*dict, Optional*) – Configuration for the database. Default to None.
- **remove_orphan_files** (*bool, Optional*) – Boolean flag to remove files on disc without a reference to the database. See *Standard tables* and *_remove_orphan_files()*. Defaults to True
- **replica_id** (*str, Optional*) – The replica identifier for the current processor.
- **create_standard_tables** (*bool, Optional*) – Boolean flag to create std tables on disk. Defaults to True When a nested steering configuration is loaded, this value can be overridden by the global create_standard_tables entry. Flat processor configurations keep the constructor value.
- **max_workers** (*int, Optional*) – Number of worker threads for parallel loops.
- **queue_size** (*int, Optional*) – Maximum size of the internal queue for the queue-based parallel loop.

- **queue_batch_size** (*int, Optional*) – Number of items processed per worker task in the queue-based parallel loop.
- **kwargs** – Keyword arguments that can be used to set processor parameters.

format_progress_message() → None

Customizes the progress message with information about the current item.

The user can overload this method in order to modify the message being displayed during the process loop with information about the current item.

The user can access the current value, its position in the looping cycle and the total number of items using *processor.Processor.item*, *processor.Processor.i_item* and *processor.Processor.n_item*.

start() → None

The start method.

The filename parser is created using the provided configuration file.

Raises
ParserConfigurationError – If the configuration file is not valid.

_filename_parser: *FilenameParser*

The filename parser instance

mafw.processor_library.sns_plotter

Module implements a Seaborn plotter processor with a mixin structure to generate seaborn plots.

This module implements the *abstract_plotter* functionalities using seaborn and pandas.

These two packages are not installed in the default installation of MAFw, unless the user decided to include the optional feature *seaborn*.

Along with the *SNSPlotter*, it includes a set of standard data retriever specific for pandas data frames.

Classes

CatPlot([x, y, hue, row, col, palette, ...])	The categorical plot mixin.
DisPlot([x, y, hue, row, col, palette, ...])	The distribution plot mixin.
FromDatasetDataRetriever([dataset_name])	A data retriever to get a dataframe from a seaborn dataset
HDFPdDataRetriever([hdf_filename, key])	Retrieve a data frame from a HDF file
LMPlot([x, y, hue, row, col, palette, ...])	The linear regression model plot mixin.
PdDataRetriever(*args, **kwargs)	The dataframe instance.
RelPlot([x, y, hue, row, col, palette, ...])	The relational plot mixin.
SNSFigurePlotter(*args, **kwargs)	Base mixin class to generate a seaborn Figure level plot
SNSPlotter(*args, **kwargs)	The Generic Plotter processor.
SQLPdDataRetriever([table_name, ...])	A specialized data retriever to get a data frame from a database table.

class mafw.processor_library.sns_plotter.**CatPlot**(*x: str | bytes | date | datetime | timedelta | bool | complex | Timestamp | Timedelta | Iterable[float | complex | int] | None = None*, *y: str | bytes | date | datetime | timedelta | bool | complex | Timestamp | Timedelta | Iterable[float | complex | int] | None = None*, *hue: str | bytes | date | datetime | timedelta | bool | complex | Timestamp | Timedelta | Iterable[float | complex | int] | None = None*, *row: str | bytes | date | datetime | timedelta | bool | complex | Timestamp | Timedelta | Iterable[float | complex | int] | None = None*, *col: str | bytes | date | datetime | timedelta | bool | complex | Timestamp | Timedelta | Iterable[float | complex | int] | None = None*, *palette: str | Sequence[tuple[float, float, float] | str | tuple[float, float, float, float] | tuple[tuple[float, float, float] | str, float] | tuple[tuple[float, float, float, float], float]] | Mapping[Any, tuple[float, float, float] | str | tuple[float, float, float, float] | tuple[tuple[float, float, float] | str, float] | tuple[tuple[float, float, float, float], float]] | None = None*, *kind: Literal['strip', 'swarm', 'box', 'violin', 'boxen', 'point', 'bar', 'count'] = 'strip'*, *legend: Literal['auto', 'brief', 'full'] | bool = 'auto'*, *native_scale: bool = False*, *plot_kws: Mapping[str, Any] | None = None*, *facet_kws: dict[str, Any] | None = None*, **args: Any*, ***kwargs: Any*)

Bases: *SNSFigurePlotter*

The categorical plot mixin.

This mixin will produce a figure level categorical plot as described here.

Constructor parameters:

Parameters

- **x** (*str | Iterable, Optional*) – The name of the x variable or an iterable containing the x values.
- **y** (*str | Iterable, Optional*) – The name of the y variable or an iterable containing the y values.
- **hue** (*str | Iterable, Optional*) – The name of the hue variable or an iterable containing the hue values.
- **row** (*str | Iterable, Optional*) – The name of the row category or an iterable containing the row values.
- **col** (*str | Iterable, Optional*) – The name of the column category or an iterable containing the column values.
- **palette** (*str, Optional*) – The colour palette to be used.
- **kind** (*str, Optional*) – The type of relational plot (scatter or line). Defaults to scatter.
- **legend** (*str | bool, Optional*) – How to draw the legend. If “brief”, numeric hue and size variables will be represented with a sample of evenly spaced values. If “full”, every group will get an entry in the legend. If “auto”, choose between brief or full representation based on number of levels. If False, no legend data is added and no legend is drawn. Defaults to auto.

- **native_scale** (*bool, Optional*) – When True, numeric or datetime values on the categorical axis will maintain their original scaling rather than being converted to fixed indices. Defaults to False.
- **plot_kws** (*dict[str, Any], Optional*) – A dictionary like list of keywords passed to the underlying seaborn.catplot.
- **facet_kws** (*dict[str, Any], Optional*) – A dictionary like list of keywords passed to the underlying seaborn.FacetGrid

plot() → None

Implements the plot method of a figure-level categorical graph.

class mafw.processor_library.sns_plotter.**DisPlot**(*x: str | bytes | date | datetime | timedelta | bool | complex | Timestamp | Timedelta | Iterable[float | complex | int] | None = None*, *y: str | bytes | date | datetime | timedelta | bool | complex | Timestamp | Timedelta | Iterable[float | complex | int] | None = None*, *hue: str | bytes | date | datetime | timedelta | bool | complex | Timestamp | Timedelta | Iterable[float | complex | int] | None = None*, *row: str | bytes | date | datetime | timedelta | bool | complex | Timestamp | Timedelta | Iterable[float | complex | int] | None = None*, *col: str | bytes | date | datetime | timedelta | bool | complex | Timestamp | Timedelta | Iterable[float | complex | int] | None = None*, *palette: str | Sequence[tuple[float, float, float] | str | tuple[float, float, float, float] | tuple[tuple[float, float, float] | str, float] | tuple[tuple[float, float, float, float], float]] | Mapping[Any, tuple[float, float, float] | str | tuple[float, float, float, float] | tuple[tuple[float, float, float] | str, float] | tuple[tuple[float, float, float, float], float]] | Colormap | None = None*, *kind: Literal['hist', 'kde', 'ecdf'] = 'hist'*, *legend: bool = True*, *rug: bool = False*, *rug_kws: dict[str, Any] | None = None*, *plot_kws: Mapping[str, Any] | None = None*, *facet_kws: dict[str, Any] | None = None*, **args: Any*, ***kwargs: Any*)

Bases: *SNSFigurePlotter*

The distribution plot mixin.

This mixin is the MAFw implementation of the seaborn displot and will produce one of the following figure level plots:

- **histplot**: a simple histogram plot
- **kdeplot**: a kernel density estimate plot
- **ecdfplot**: an empirical cumulative distribution functions plot
- **rugplot**: a plot of the marginal distributions as ticks.

Constructor parameters:

Parameters

- **x** (*str | Iterable, Optional*) – The name of the x variable or an iterable containing the x values.
- **y** (*str | Iterable, Optional*) – The name of the y variable or an iterable containing the y values.

- **hue** (*str | Iterable, Optional*) – The name of the hue variable or an iterable containing the hue values.
- **row** (*str | Iterable, Optional*) – The name of the row category or an iterable containing the row values.
- **col** (*str | Iterable, Optional*) – The name of the column category or an iterable containing the column values.
- **palette** (*str | Colormap, Optional*) – The colour palette to be used.
- **kind** (*str, Optional*) – The type of distribution plot (hist, kde or ecdf). Defaults to hist.
- **legend** (*bool, Optional*) – If false, suppress the legend for the semantic variables. Defaults to True.
- **rug** (*bool, Optional*) – If true, show each observation with marginal ticks. Defaults to False.
- **rug_kws** (*Mapping[str, Any], Optional*) – Parameters to control the appearance of the rug plot.
- **plot_kws** (*Mapping[str, Any], Optional*) – Parameters passed to the underlying plotting object.
- **facet_kws** (*Mapping[str, Any], Optional*) – Parameters passed to the facet grid object.

plot() → None

Implements the plot method for a figure-level distribution graph

class mafw.processor_library.sns_plotter.**FromDatasetDataRetriever**(*dataset_name: str | None = None*, **args: Any*, ***kwargs: Any*)

Bases: *PdDataRetriever*

A data retriever to get a dataframe from a seaborn dataset

The dataframe instance. It will be filled for the main class

_attributes_valid() → bool

Checks if the attributes of the mixin are all valid

get_data_frame() → None

Gets the data frame from the standard seaborn datasets

class mafw.processor_library.sns_plotter.**HDFPdDataRetriever**(*hdf_filename: str | Path | None = None*, *key: str | None = None*, **args: Any*, ***kwargs: Any*)

Bases: *DataRetriever*

Retrieve a data frame from a HDF file

This data retriever is getting a dataframe from a HDF file provided the filename and the object key.

Constructor parameters:

Parameters

- **hdf_filename** (*str | Path, Optional*) – The filename of the HDF file
- **key** (*str, Optional*) – The key of the HDF store with the dataframe

get_data_frame() → None

Retrieve the dataframe from a HDF file

Raises

PlotterMixinNotInitialized – if some of the required attributes are not initialised or invalid.

patch_data_frame() → None

The mixin implementation of the shared method with the base class

class mafw.processor_library.sns_plotter.**LMPlot**(*x: str | None = None*, *y: str | None = None*, *hue: str | None = None*, *row: str | None = None*, *col: str | None = None*, *palette: str | Sequence[tuple[float, float, float]] | str | tuple[float, float, float, float] | tuple[tuple[float, float, float] | str, float] | tuple[tuple[float, float, float, float], float]] | Mapping[Any, tuple[float, float, float] | str | tuple[float, float, float, float] | tuple[tuple[float, float, float] | str, float] | tuple[tuple[float, float, float, float], float]] | None = None*, *legend: bool = True*, *scatter_kws: dict[str, Any] | None = None*, *line_kws: dict[str, Any] | None = None*, *facet_kws: dict[str, Any] | None = None*, **args: Any*, ***kwargs: Any*)

Bases: *SNSFigurePlotter*

The linear regression model plot mixin.

This mixin will produce a figure level regression model as described here

Constructor parameters:

Parameters

- **x** (*str, Optional*) – The name of the x variable or an iterable containing the x values.
- **y** (*str, Optional*) – The name of the y variable or an iterable containing the y values.
- **hue** (*str, Optional*) – The name of the hue variable or an iterable containing the hue values.
- **row** (*str, Optional*) – The name of the row category or an iterable containing the row values.
- **col** (*str, Optional*) – The name of the column category or an iterable containing the column values.
- **palette** (*str, Optional*) – The colour palette to be used.
- **legend** (*bool, Optional*) – If True and there is a hue variable, add a legend.
- **scatter_kws** (*dict[str, Any], Optional*) – A dictionary like list of keywords passed to the underlying scatter.
- **scatter_kws** – A dictionary like list of keywords passed to the underlying scatter.
- **line_kws** (*dict[str, Any], Optional*) – A dictionary like list of keywords passed to the underlying plot.
- **facet_kws** (*dict[str, Any], Optional*) – A dictionary like list of keywords passed to the underlying seaborn.FacetGrid

plot() → None

Implements the plot method for a figure-level regression model.

class mafw.processor_library.sns_plotter.**PdDataRetriever**(**args: Any*, ***kwargs: Any*)

Bases: *DataRetriever*

The dataframe instance. It will be filled for the main class

get_data_frame() → None

The mixin implementation of the shared method with the base class

patch_data_frame() → None

The mixin implementation of the shared method with the base class

class mafw.processor_library.sns_plotter.**RelPlot**(*x: str | bytes | date | datetime | timedelta | bool | complex | Timestamp | Timedelta | Iterable[float | complex | int] | None = None*, *y: str | bytes | date | datetime | timedelta | bool | complex | Timestamp | Timedelta | Iterable[float | complex | int] | None = None*, *hue: str | bytes | date | datetime | timedelta | bool | complex | Timestamp | Timedelta | Iterable[float | complex | int] | None = None*, *row: str | bytes | date | datetime | timedelta | bool | complex | Timestamp | Timedelta | Iterable[float | complex | int] | None = None*, *col: str | bytes | date | datetime | timedelta | bool | complex | Timestamp | Timedelta | Iterable[float | complex | int] | None = None*, *palette: str | Sequence[tuple[float, float, float] | str | tuple[float, float, float, float] | tuple[tuple[float, float, float] | str, float] | tuple[tuple[float, float, float, float], float]] | Mapping[Any, tuple[float, float, float] | str | tuple[float, float, float, float] | tuple[tuple[float, float, float] | str, float] | tuple[tuple[float, float, float, float], float]] | Colormap | None = None*, *kind: Literal['scatter', 'line'] = 'scatter'*, *legend: Literal['auto', 'brief', 'full'] | bool = 'auto'*, *plot_kws: Mapping[str, Any] | None = None*, *facet_kws: dict[str, Any] | None = None*, **args: Any*, ***kwargs: Any*)

Bases: *SNSFigurePlotter*

The relational plot mixin.

This mixin will produce either a scatter or a line figure level plot.

The full documentation of the relplot object can be read at this link.

Constructor parameters:

Parameters

- **x** (*str | Iterable, Optional*) – The name of the x variable or an iterable containing the x values.
- **y** (*str | Iterable, Optional*) – The name of the y variable or an iterable containing the y values.
- **hue** (*str | Iterable, Optional*) – The name of the hue variable or an iterable containing the hue values.
- **row** (*str | Iterable, Optional*) – The name of the row category or an iterable containing the row values.
- **col** (*str | Iterable, Optional*) – The name of the column category or an iterable containing the column values.
- **palette** (*str | Colormap, Optional*) – The colour palette to be used.
- **kind** (*str, Optional*) – The type of relational plot (scatter or line). Defaults to scatter.

- **legend** (*str | bool, Optional*) – How to draw the legend. If "brief", numeric hue and size variables will be represented with a sample of evenly spaced values. If "full", every group will get an entry in the legend. If "auto", choose between brief or full representation based on number of levels. If False, no legend data is added and no legend is drawn. Defaults to auto.
- **plot_kws** (*dict[str, Any], Optional*) – A dictionary like list of keywords passed to the underlying seaborn.relplot.
- **facet_kws** (*dict[str, Any], Optional*) – A dictionary like list of keywords passed to the underlying seaborn.FacetGrid

plot() → None

Implements the plot method of a figure-level relational graph.

class mafw.processor_library.sns_plotter.**SNSFigurePlotter**(**args: Any*, ***kwargs: Any*)

Bases: *FigurePlotter*

Base mixin class to generate a seaborn Figure level plot

data_frame: DataFrame

The dataframe instance shared with the main class

facet_grid: FacetGrid

The facet grid instance shared with the main class

class mafw.processor_library.sns_plotter.**SNSPlotter**(**args: Any*, ***kwargs: Any*)

Bases: *GenericPlotter*

The Generic Plotter processor.

This is a subclass of a Processor with advanced functionality to fetch data in the form of a dataframe and to produce plots.

The key difference with respect to a normal processor is it *process()* method that has been already implemented as follows:

```
def process(self) -> None:
    """
    Specific implementation of the process method for the Seaborn plotter.

    It is almost the same as the GenericProcessor, with the addition that all␣
↪open pyplot figures are closed
    after the process is finished.

    This part cannot be moved upward to the :class:`~.GenericPlotter` because the␣
↪user might have selected
    another plotting library different from :link:`matplotlib`.
    """
    super().process()
    if not self.is_data_frame_empty():
        plt.close('all')
```

This actually means that when you are subclassing a SNSPlotter you do not have to implement the process method as you would do for a normal Processor, but you will have to implement the following methods:

- *in_loop_customization()*.

  The processor execution workflow (LoopType) can be any of the available, so actually the process method might be invoked only once, or multiple times inside a loop structure (for or while). If the execution is cyclic, then you may want to have the possibility to do some customisation for each iteration, for example, changing the plot title, or modifying the data selection, or the filename where the plots will be saved.

You can use this method also in case of a single loop processor, in this case you will not have access to the loop parameters.

- *get_data_frame()*.

  This method has the task to get the data to be plotted in the form of a pandas DataFrame. The processor has the *data_frame* attribute where the data will be stored to make them accessible from all other methods.

- *patch_data_frame()*.

  A convenient method to apply data frame manipulation to the data just retrieved.

- *plot()*.

  This method is where the actual plotting occurs. Use the *data_frame* to plot the quantities you want.

- *customize_plot()*.

  This method can be optionally used to customize the appearance of the facet grid produced by the *plot()* method. It is particularly useful when the user is mixing this class with one of the *FigurePlotter* mixin, thus not having direct access to the plot method.

- *save()*.

  This method is where the produced plot is saved in a file. Remember to append the output file name to the *list of produced outputs* so that the *_update_plotter_db()* method will automatically store this file in the database during the *finish()* execution.

- *update_db()*.

  If the user wants to update a specific table in the database, they can use this method.

  It is worth reminding that all plotters are saving all generated files in the standard table PlotterOutput. This is automatically done by the *_update_plotter_db()* method that is called in the *finish()* method.

You do not need to overload the *slice_data_frame()* nor the *group_and_aggregate_data_frame()* methods, but you can simply use them by setting the *slicing_dict* and the *grouping_columns* and the *aggregation_functions*.

The processor comes with two processors parameters that can be used by user-defined subclasses:

1. The output_folder that is the path where the output file will be saved
2. The force_replot flag to be used whether the user wants the plot to be regenerated even if the output file already exists.

The default value of these parameters can be changed using the *processor.Processor.new_defaults* dictionary as shown in *this example*.

Processor parameters

- **force_replot**: Whether to force re-plotting even if the output file already exists (default: False)
- **output_folder**: The path where the output file will be saved (default: PosixPath('/tmp/mafw-docs-kpuvl5nc/v2.3.0_pdf'))

Constructor parameters:

Parameters

- **slicing_dict** (*dict[str, Any], Optional*) – A dictionary with key, value pairs to slice the input data frame before the plotting occurs.
- **grouping_columns** (*list[str], Optional*) – A list of columns for the groupby operation on the data frame.
- **aggregation_functions** (*list[str | Callable[[Any], Any], Optional*) – A list of functions for the aggregation on the grouped data frame.

- **matplotlib_backend** (*str, Optional*) – The name of the matplotlib backend to be used. Defaults to 'Agg'
- **output_folder** (*Path, Optional*) – The path where the output file will be saved
- **force_replot** (*bool, Optional*) – Whether to force re-plotting even if the output file already exists.

get_data_frame() → None

Specific implementation of the get data frame for the Seaborn plotter.

It must be overloaded.

The method is **NOT** returning the data_frame, but in your implementation you need to assign the data frame to the class *data_frame* attribute.

group_and_aggregate_data_frame() → None

Performs groupyby and aggregation of the data frame.

If the user provided some *grouping columns* and *aggregation functions* then the *group_and_aggregate_data_frame()* is invoked accordingly.

The user can update the values of those attributes during each cycle iteration within the implementation of the *in_loop_customization()*.

See also

This method is simply invoking the *group_and_aggregate_data_frame()* function from the *pandas_tools*.

is_data_frame_empty() → bool

Check if the data frame is empty

process() → None

Specific implementation of the process method for the Seaborn plotter.

It is almost the same as the GenericProcessor, with the addition that all open pyplot figures are closed after the process is finished.

This part cannot be moved upward to the *GenericPlotter* because the user might have selected another plotting library different from matplotlib.

slice_data_frame() → None

Perform data frame slicing

The user can set some slicing criteria in the *slicing_dict* to select some specific data subset. The values of the slicing dict can be changed during each iteration within the implementation of the *in_loop_customization()*.

See also

This method is simply invoking the *slice_data_frame()* function from the *pandas_tools*.

start() → None

Overload of the start method.

The *SNSPlotter* is overloading the *start()* in order to allow the user to change the matplotlib backend.

The user can selected which backend to use either directly in the class constructor or assign it to the class attribute *matplotlib_backend*.

_methods_to_be_checked_for_super: tuple[str, ...] = ('start', 'finish', 'patch_data_frame')

Methods that must invoke super when overridden.

aggregation_functions: Iterable[str | Callable[[Any], Any]] | None

The list of aggregation functions to be applied to the grouped dataframe

data_frame: pd.DataFrame

The pandas DataFrame containing the data to be plotted.

facet_grid: sns.FacetGrid | None

The reference to the facet grid.

grouping_columns: Iterable[str] | None

The list of columns for grouping the data frame

matplotlib_backend: str

The backend to be used for matplotlib.

output_filename_list: list[Path]

The list of produced filenames.

slicing_dict: MutableMapping[str, Any] | None

The dictionary for slicing the input data frame

class mafw.processor_library.sns_plotter.**SQLPdDataRetriever**(*table_name: str | None = None*, *required_cols: Iterable[str] | str | None = None*, *where_clause: str | None = None*, **args: Any*, ***kwargs: Any*)

Bases: *PdDataRetriever*

A specialized data retriever to get a data frame from a database table.

The idea is to implement an interface to the pandas `read_sql`. The user has to provide the *table name*, the *the list of required columns* and an optional *where clause*.

Constructor parameters:

Parameters

- **table_name** (*str, Optional*) – The name of the table from where to get the data
- **required_cols** (*Iterable[str] | str | None, Optional*) – A list of columns to be selected from the table and transferred as column in the dataframe.
- **where_clause** (*str, Optional*) – The where clause used in the select SQL statement. If None is provided, then all rows will be selected.

_attributes_valid() → bool

Check if all required parameters are provided and valid.

get_data_frame() → None

Retrieve the dataframe from a database table.

Raises

PlotterMixinNotInitialized – If some of the required attributes are missing.

database: Database

The database instance. It comes from the main class

required_columns: Iterable[str]

The iterable of columns.

Those are the column names to be selected from the *table_name* and included in the dataframe.

table_name: str

The table from where the data should be taken.

where_clause: str

The where clause of the SQL statement

14.1.15 mafw.runner

Provides a container to run configurable and modular analytical tasks.

Classes

MAFwApplication([steering_file, ...])	The MAFw Application.

class mafw.runner.**MAFwApplication**(*steering_file: Path | str | None = None*, *user_interface:* UserInterfaceBase | *type[*UserInterfaceBase*]* | *str | None = None*, *plugin_manager:* MAFwPluginManager | *None = None*)

Bases: `object`

The MAFw Application.

This class takes care of reading a steering file and from the information retrieved construct a *ProcessorList* from the processor listed there and execute it.

It is very practical because any combination of processors can be run without having to write dedicated scripts but simply modifying a steering file.

The application will search for processors not only among the ones available in the MAFw library, but also in all other packages exposing processors via the *plugin mechanism*.

All parameters in the constructor are optional.

An instance can be created also without the *steering_file*, but such an instance cannot be executed. The steering file can be provided in a later stage via the *init()* method or directly to the *run()* method.

The user interface can be either provided directly in the constructor, or it will be taken from the steering file. In the worst case, the fallback *ConsoleInterface* will be used.

The plugin manager, if not provided, the global plugin manager will be retrieved from the *get_plugin_manager()*.

A simple example is provided here below:

Listing 14.1: Creation and execution of a MAFwApplication

```
import logging
from pathlib import Path

from mafw.runner import MAFwApplication

log = logging.getLogger(__name__)

# put here your steering file
steering_file = Path('path_to_my_steering_file.toml')

try:
    # create the app
    app = MAFwApplication(steering_file)

    # run it!
    app.run()
```

(continues on next page)

(continued from previous page)

```
except Exception as e:
    log.error('An error occurred!')
    log.exception(e)
```

Constructor parameters:

Parameters

- **steering_file** (*Path | str, Optional*) – The path to the steering file.
- **user_interface** (*UserInterfaceBase | type[UserInterfaceBase] | str, Optional*) – The user interface to be used by the application.
- **plugin_manager** (*PluginManager, Optional*) – The plugin manager.

_expand_processors_to_run(*items: list[str]*, *config: dict[str, Any]*, *plugins:* LoadedPlugins, *seen: set[str] | None = None*) → *ProcessorList*

Constructs a *processor.ProcessorList* from a list of processor names or groups.

This method recursively expands a list of processor names or group definitions into a *ProcessorList* object, which can be executed. It handles three cases: direct processor class matches, group definitions in the configuration, and unknown entries.

Added in version v2.0.0.

Parameters

- **items** (*list[str]*) – A list of processor names or group identifiers to be expanded.
- **config** (*dict[str, Any]*) – The configuration dictionary containing processor and group definitions.
- **plugins** (LoadedPlugins) – The loaded plugins containing available processors and groups.
- **seen** (*set[str] | None*) – A set of group names that have already been processed to detect cyclic definitions.

Returns

A *ProcessorList* containing the expanded processors ready for execution.

Return type

processor.ProcessorList

Raises

- ***UnknownProcessorGroup*** – If a group is defined without processors or if a cyclic group definition is detected.
- ***UnknownProcessor*** – If a processor name is not recognized in the plugin library.

get_user_interface(*user_interface: str*) → *UserInterfaceBase*

Retrieves the user interface from the loaded plugins.

User interfaces are exposed via the plugin manager.

If the requested *user_interface* is not available, then the fallback console interface is used.

Parameters

user_interface (*str*) – The name of the user interface to be used. Normally rich or console.

init(*steering_file: Path | str*) → None

Initializes the application.

This method is normally automatically invoked by the class constructor. It can be called in a later moment to force the parsing of the provided steering file.

Parameters
steering_file (*Path | str*) – The path to the steering file.

run(*steering_file: Path | str | None = None*) → *ProcessorExitStatus*

Runs the application.

This method builds the *ProcessorList* with the processors listed in the steering file and launches its execution.

A steering file can be provided at this stage if it was not done before.

Changed in version v2.0.0: Refactor to accept replica names in the processor to run list.

Parameters
steering_file (*Path | str, Optional*) – The steering file. Defaults to None.

Raises

- ***RunnerNotInitialized*** – if the application has not been initialized. Very likely a steering file was never provided.
- ***UnknownProcessor*** – if a processor listed in the steering file is not available in the plugin library.

exit_status
the exit status of the application

name
the name of the application instance

plugin_manager
the plugin manager of the application instance

14.1.16 mafw.scripts

Executables.

This package includes the executables necessary to run MAFw.

Modules

mafw_exe	The execution framework.

mafw.scripts.mafw_exe

The execution framework.

This module provides the run functionality to the whole library.

It is heavily relying on `click` for the generation of commands, options, and arguments.

mafw

The Modular Analysis Framework execution.

This is the command line interface where you can configure and launch your analysis tasks.

More information on our documentation page.

param ctx
The click context.

type ctx
click.core.Context

param log_level
The logging level as a string. Choice from debug, info, warning, error and critical.

type log_level
str

param ui
The user interface as a string. Choice from console and rich.

type ui
str

param debug
Flag to show debug information about exception.

type debug
bool

param no_banner
Flag to disable the welcome banner.

type no_banner
bool

Usage

```
mafw [OPTIONS] [COMMAND] [ARGS]...
```

Options

`--log-level` <log_level>
Log level

Default
'info'

Options
debug | info | warning | error | critical

`--ui` <ui>
The user interface

Default
'rich'

Options
console | rich

`-D, --debug`
Show debug information about errors

`--no-banner`
Disable the welcome banner

`-v, --version`
Show the version and exit.

completion

Manage shell completion for the `mafw` command.

The completion workflow installs the Click-generated shell code into the active virtual environment, updates the activation script so completion is loaded automatically, and exposes a `show` helper for direct evaluation.

Added in version v2.2.

param ctx
The click context.

type ctx
click.core.Context

Usage

```
mafw completion [OPTIONS] COMMAND [ARGS]...
```

install

Install the `mafw` shell completion script.

When `--shell` is omitted, the command guesses the shell from `$SHELL`. The generated Click completion script is stored in the active virtual environment and the activation script is updated so that future shell sessions load completion automatically.

Added in version v2.2.

param ctx
The click context.

type ctx
click.core.Context

param shell
Target shell selector.

type shell
str

param force
Reinstall completion even if already loaded.

type force
bool

Usage

```
mafw completion install [OPTIONS]
```

Options

`-s, --shell` <shell>
Target shell for completion installation

Default
`'auto'`

Options
auto | bash | zsh | fish

`-F, --force`
Reinstall completion even if already loaded

show

Display the Click completion script on standard output.

The output is intentionally clean so it can be used with `eval`:

```
eval "$(mafw completion show)"
```

Added in version v2.2.

param ctx
The click context.

type ctx
click.core.Context

param shell
Target shell selector.

type shell
str

Usage

```
mafw completion show [OPTIONS]
```

Options

-s, --shell <shell>
Target shell for completion output

Default
'auto'

Options
auto | bash | zsh | fish

uninstall

Remove the installed `mafw` shell completion files.

This command removes generated completion files from `share/mafw` and strips the marker block from the activation scripts.

Added in version v2.2.

param ctx
The click context.

type ctx
click.core.Context

Usage

```
mafw completion uninstall [OPTIONS]
```

db

Advanced database commands.

The db group of commands offers a set of useful database operations. Invoke the help option of each command for more details.

param ctx
The click context.

type ctx
click.core.Context

Usage

```
mafw db [OPTIONS] COMMAND [ARGS]...
```

wizard

Reflect an existing DB into a python module.

mafw db wizard [Options] Database

Database Name of the Database to be reflected.

About connection options (user / host / port):

That information will be used only in case you are trying to access a network database (MySQL or PostgreSQL). In case of Sqlite, the parameters will be discarded.

About passwords:

If you need to specify a password to connect to the DB server, just add –password in the command line without typing your password as clear text. You will be prompted to insert the password with hidden characters at the start of the processor.

About engines:

The full list of supported engines is provided in the option below. If you do not specify any engine and the database is actually an existing filename, then engine is set to Sqlite, otherwise to postgresql.

param database
The name of the database.

type database
str

param schema
The database schema to be reflected.

type schema
str

param engine
The database engine. A selection of possible values is provided in the script help.

type engine
str

param password
The password for the DB connection. Not used in case of Sqlite.

type password
str

param user
The username for the DB connection. Not used in case of Sqlite.

type user
str

param port
The port number of the database server. Not used in case of Sqlite.

type port
int

param host
The database hostname. Not used in case of Sqlite.

type host
str

param output_file
The filename for the output python module.

type output_file
click.Path | pathlib.Path | str

param snake_case
Flag to select snake_case convention for table and field names, or all small letter formatting.

type snake_case
bool

param ignore_unknown
Flag to ignore unknown fields. If False, an unknown field will be labelled with UnknownField.

type ignore_unknown
bool

param with_views
Flag to include views in the reflected elements.

type with_views
bool

param preserve_order
Flag to select if table fields should be reflected in the original order (True) or in alphabetical order (False)

type preserve_order
bool

param tables
A tuple containing a selection of table names to be reflected.

type tables
tuple[str, ...]

param overwrite
Flag to overwrite the output file if exists. If False and the output file already exists, the user can decide what to do.

type overwrite
bool

param ctx
The click context, that includes the original object with global options.

type ctx
click.core.Context

return
The script return value

Usage

```
mafw db wizard [OPTIONS] DATABASE
```

Options

-o, --output-file <output_file>
The name of the output file with the reflected model.

-s, --schema <schema>
The name of the DB schema

-t, --tables <tables>
Generate model for selected tables. Multiple option possible.

--overwrite, --no-overwrite
Overwrite output file if already exists.

--preserve-order, --no-preserve-order
Preserve column order.

--with-views, --without-views
Include also database views.

--ignore-unknown, --no-ignore-unknown
Ignore unknown fields.

--snake-case, --no-snake-case
Use snake case for table and field names.

--host <host>
Hostname for the DB server.

-p, --port <port>
Port number for the DB server.

-u, --user, --username <user>
Username for the connection to the DB server.

--password <password>
Insert password when prompted

-e, --engine <engine>
The DB engine

Options
cockroach | cockroachdb | crdb | mysql | mysqldb | postgres | postgresql | sqlite | sqlite3

Arguments

DATABASE
Required argument

list

Display the list of available processors.

This command will retrieve all available processors via the plugin manager. Both internal and external processors will be listed if the ext-plugins option is passed.

Usage

```
mafw list [OPTIONS]
```

run

Runs a steering file.

STEERING_FILE A path to the steering file to execute.

param obj
The context object being passed from the main command.

type obj
dict

param steering_file
The path to the output steering file.

type steering_file
Path

Usage

```
mafw run [OPTIONS] STEERING_FILE
```

Arguments

STEERING_FILE
Required argument

steering

Generates a steering file with the default parameters of all available processors.

STEERING_FILE A path to the steering file to execute.

The user must modify the generated steering file to ensure it can be executed using the run command.

param obj
The context object being passed from the main command.

type obj
dict

param show
Display the steering file in the console after the generation. Defaults to False.

type show
bool

param ext_plugins
Extend the search for processor to external libraries.

type ext_plugins
bool

param open_editor
Open a text editor after the generation to allow direct editing.

type open_editor
bool

param steering_file
The steering file path.

type steering_file
Path

param db_engine
The name of the db engine.

type db_engine
str

param db_url
The URL of the database.

type db_url
str

Usage

```
mafw steering [OPTIONS] STEERING_FILE
```

Options

--show, --no-show

Display the generated steering file on console

--ext-plugins, --no-ext-plugin

Load external plugins

--open-editor, --no-open-editor

Open the file in your editor.

--db-engine <db_engine>

Select a DB engine

Options
sqlite | mysql | postgresql

--db-url <db_url>

URL to the DB

Arguments

STEERING_FILE

Required argument

Functions

custom_formatwarning(message, category, ...)	Return the pure message of the warning.
display_exception(exception[, show_traceback])	Display exception information with optional debug details.
is_bugged_warning_capture_version()	Check if the Python version has the warning capture bug in free-threading mode.
logger_setup(level, ui, tracebacks)	Set up the logger.
print_banner(ctx)	Print the welcome banner only once, only for rich UI, and only if not disabled.

Classes

MAFwGroup([name, commands, ...])	Click group with abbreviation and MAFw-specific exit handling.
ReturnValue(*values)	Enumerator to handle the script return value.

class mafw.scripts.mafw_exe.**MAFwGroup**(*name: str | None = None*, *commands: MutableMapping[str, Command] | Sequence[Command] | None = None*, *invoke_without_command: bool = False*, *no_args_is_help: bool | None = None*, *subcommand_metavar: str | None = None*, *chain: bool = False*, *result_callback: Callable[[...], Any] | None = None*, ***kwargs: Any*)

Bases: *AbbreviateGroup*

Click group with abbreviation and MAFw-specific exit handling.

group_class

alias of *AbbreviateGroup*

_detect_command_name(*args: list[str]*) → str | None

Return the first subcommand token while skipping root option values.

invoke(*ctx: Context*) → Any

Invoke the command and normalize Click exceptions to MAFw exits.

main(**args: Any*, ***kwargs: Any*) → None

Run the CLI and convert returned statuses to process exit codes.

parse_args(*ctx: Context*, *args: list[str]*) → list[str]

Parse arguments and record the selected root command before invoke.

class mafw.scripts.mafw_exe.**ReturnValue**(**values*)

Bases: IntEnum

Enumerator to handle the script return value.

Error = 1

Generic error

OK = 0

No error

mafw.scripts.mafw_exe.**custom_formatwarning**(*message: Warning | str*, *category: type[Warning]*, *filename: str*, *lineno: int*, *line: str | None = None*) → str

Return the pure message of the warning.

mafw.scripts.mafw_exe.**display_exception**(*exception: Exception*, *show_traceback: bool = False*) → None

Display exception information with optional debug details.

This function logs exception information at the critical level. When show_traceback is enabled, it logs the full exception including traceback information. Otherwise, it logs a simplified message directing users to enable debug mode for more details.

Parameters

- **exception** (*Exception*) – The exception to be displayed and logged.
- **show_traceback** (*bool*) – Flag indicating whether to show detailed traceback information. Defaults to False

mafw.scripts.mafw_exe.**is_bugged_warning_capture_version**() → bool

Check if the Python version has the warning capture bug in free-threading mode.

Python 3.14.0 through 3.14.3 (free-threading builds only) have a bug where `logging.captureWarnings(True)` can crash or misbehave. This function returns True when the current interpreter is affected.

Note

See https://github.com/python/cpython/pull/146374 for details.

Todo

Remove this workaround when Python 3.14 is no longer a supported version (expected late 2030).

Returns
True if the interpreter is affected by the warning capture bug, False otherwise.

Return type
bool

`mafw.scripts.mafw_exe.logger_setup`(*level: str*, *ui: str*, *tracebacks: bool*) → None

Set up the logger.

This function is actually configuring the root logger level from the command line options and it attaches either a RichHandler or a StreamHandler depending on the user interface type.

The *tracebacks* flag is used only by the RichHandler. Printing the tracebacks is rather useful when debugging the code, but it could be detrimental for final users. In normal circumstances, tracebacks is set to False, and is turned on when the debug flag is activated.

Parameters

- **level** (*str*) – Logging level as a string.
- **ui** (*str*) – User interface as a string ('rich' or 'console').
- **tracebacks** – Enable/disable the logging of exception tracebacks.

`mafw.scripts.mafw_exe.print_banner`(*ctx: Context*) → None

Print the welcome banner only once, only for rich UI, and only if not disabled.

> **Note**
>
> ctx.obj is not yet populated when Group.invoke() is called, but ctx.params contains parsed options for the current group. We therefore inspect the root context params.

14.1.17 mafw.steering

Steering builder tools for MAFw.

Author
Bulgheroni Antonio

Description
Editable data model, validation, and TOML serialization helpers for steering files.

Modules

builder	Editable steering configuration builder independent from the execution engine.
models	Domain models for steering metadata construction.
serializer	Serialize steering metadata into TOML documents via tomlkit.
validation	Structural validation helpers for steering metadata.

mafw.steering.builder

Editable steering configuration builder independent from the execution engine.

Author
Bulgheroni Antonio

Description
Provides helpers to construct steering metadata without running processors.

Classes

SteeringBuilder()	Editable domain model for building MAFw steering files.
ValidationLevel(*values)	Validation tiers that can be requested from the steering builder.

class mafw.steering.builder.**SteeringBuilder**

Bases: object

Editable domain model for building MAFw steering files.

classmethod from_toml(*path: Path | str*) → *SteeringBuilder*

Create a builder from an existing steering file while keeping TOML metadata.

classmethod from_toml_text(*text: str*) → *SteeringBuilder*

Create a builder from TOML text while keeping TOML metadata.

add_group(*name: str*, *processors: Iterable[str]*, *description: str | None = None*) → None

Register a processor group.

add_processor(*base_name: str*, *replica: str | None = None*) → None

Add a processor reference to the global processors_to_run list.

add_replica(*base_name: str*, *replica: str*) → None

Create a replica entry without touching the base configuration.

clear_parameters(*processor_full_name: str*) → None

Clear every parameter override for a processor.

disable_db_configuration() → None

Prevent the DBConfiguration section from being emitted.

enable_db_configuration() → None

Ensure the DBConfiguration section will be serialized.

get_group(*name: str*) → *GroupConfig*

Return the stored configuration for a group section.

get_processor_config(*full_name: str*) → *ProcessorConfig*

Return the stored configuration for a processor or replica.

is_db_configuration_enabled() → bool

Return whether the DBConfiguration section should be present.

list_groups() → list[str]

Return every group section name currently configured.

list_processors() → list[str]

Return every processor section name currently configured.

remove_filter(*processor_full_name: str*, *model_name: str*) → None

Remove a filter model definition.

remove_group(*name: str*) → None

Delete a group by name.

remove_parameter(*processor_full_name: str*, *key: str*) → None

Remove a processor parameter override if present.

remove_processor(*full_name: str*) → None

Remove a processor or replica from the run list.

set_analysis_description(*description: str | None*) → None

Set `analysis_description`.

set_analysis_name(*name: str | None*) → None

Set `analysis_name`.

set_create_standard_tables(*value: bool | None*) → None

Set `create_standard_tables`.

set_db_attribute(*key: str*, *value: Any | None*) → None

Store a generic key/value pair inside DBConfiguration.

set_db_authentication(*authentication: Mapping[str, Any] | None*) → None

Set the database authentication mapping.

set_db_parameters(*parameters: Mapping[str, Mapping[str, Any]] | None*) → None

Set the backend-specific database parameters mapping.

set_db_pragmas(*pragmas: dict[str, Any]*) → None

Set database pragmas.

set_db_url(*url: str | None*) → None

Override the database URL.

set_default() → None

Initialize globals, database, and UI defaults for a fresh builder.

set_filter_conditionals(*processor_full_name: str*, *model_name: str*, *conditionals: list[dict[str, Any]] | None*) → None

Assign `__conditional__` blocks to a filter model.

set_filter_config(*processor_full_name: str*, *model_name: str*, *config: dict[str, Any]*) → None

Replace the configuration for a given filter model.

set_filter_field(*processor_full_name: str*, *model_name: str*, *field: str*, *value: Any*) → None

Set or update a single field within a filter model.

set_filter_logic(*processor_full_name: str*, *logic: str | None*) → None

Set the global `__logic__` string for the processor filters.

set_new_only(*value: bool | None*) → None

Set the top-level `new_only` flag.

set_parameter(*processor_full_name: str*, *key: str*, *value: Any*) → None

Set a processor parameter.

set_processor_filters(*processor_full_name: str*, *filters: dict[str, list[FilterConfig]]*, *logic: str | None*) → None

Update the filters and logic for a given processor.

set_processor_new_only(*processor_full_name: str*, *new_only: bool | None*) → None

Explicitly set `__new_only__` for a processor or replica.

set_processors_to_run(*processors: Iterable[str]*) → None

Overwrite the processors_to_run list.

set_replica_inheritance(*replica_full_name: str*, *inheritance: bool | None*) → None

Toggle the inheritance behaviour for a replica.

set_ui_interface(*interface: str*) → None

Pick the interface used by `UserInterface`.

to_config_dict() → dict[str, Any]

Return a plain dictionary representing the steering configuration.

to_document(*, *validation_level:* ValidationLevel | *None* = *None*) → TOMLDocument

Serialize the builder state into a TOML document.

validate(*validation_level:* ValidationLevel = *ValidationLevel.SEMANTIC*) → list[*ValidationIssue*]

Run steering validation at the requested level and report every issue.

write(*path: Path | str*, *, *validation_level:* ValidationLevel | *None* = *None*) → None

Dump the builder to disk.

property document: TOMLDocument | None

Return the parsed TOML document this builder originated from.

property extra_globals: dict[str, Any]

Return extra top-level globals preserved from the steering file.

class mafw.steering.builder.**ValidationLevel**(**values*)

Bases: Enum

Validation tiers that can be requested from the steering builder.

mafw.steering.models

Domain models for steering metadata construction.

Author

Bulgheroni Antonio

Description

Simple dataclasses that hold editable steering metadata without execution logic.

Classes

Condition(operator, value[, is_implicit, ...])	A single filter condition.
ConditionalFilterConfig(name[, enabled, ...])	Conditional filter applied to a model.
DBConfiguration(url, pragmas, ...)	Database section content mirrored from the steering file.
FieldFilterConfig(name, enabled, model, ...)	Filter applied to a single field of a specific model.
FieldInfo(name, type[, help_text])	Metadata about a model field.
FilterConfig(name[, enabled, model])	Base configuration for a filter.
FilterKind(*values)	The kind of filter configuration.
GlobalSettings(processors_to_run, ...)	Top-level globals that appear outside any processor or group table.
GroupConfig(name, processors, description, ...)	Configuration for a logical processor group section.
ModelFilterConfig(name, enabled, model, ...)	Filter applied to a specific database model.
ModelInfo(field_info, ...)	Metadata about a database model.
ParameterConfig(name[, value, default, ...])	Configuration and status for a single processor parameter.
ParameterSchemaStatus(*values)	Schema reconciliation status for a processor parameter.
ParameterSource(*values)	Origin of a processor parameter value in the GUI pipeline.
ProcessorConfig(name, parameters, ...)	Configuration store for an individual processor or replica section.
ProcessorRef(base_name[, replica])	Reference to a processor entry used in processors_to_run lists.
ProcessorSchemaStatus(*values)	Schema reconciliation status for a processor.

continues on next page

Table 14.136 – continued from previous page

UIConfiguration([interface])	User-interface section content mirrored from the steering file.

class mafw.steering.models.**Condition**(*operator: str | LogicalOp*, *value: Any*, *is_implicit: bool = True*, *is_valid: bool = True*)

Bases: `object`

A single filter condition.

class mafw.steering.models.**ConditionalFilterConfig**(*name: str*, *enabled: bool = True*, *model: str = ''*, *condition: Condition | None = None*, *then_clause: Condition | None = None*, *else_clause: Condition | None = None*, *condition_field: str = ''*, *then_field: str = ''*, *else_field: str | None = None*, *auto_named: bool = False*)

Bases: *FilterConfig*

Conditional filter applied to a model.

Semantics:
IF condition THEN then_clause ELSE else_clause

fill_model_info() → None

Populate model_info by inspecting the database model.

is_complete() → bool

Return whether every active clause of the conditional filter is valid.

class mafw.steering.models.**DBConfiguration**(*url: str | None = None*, *pragmas: dict[str, ~typing.Any] = <factory>*, *authentication: dict[str, ~typing.Any] = <factory>*, *parameters: dict[str, dict[str, ~typing.Any]] = <factory>*, *attributes: dict[str, ~typing.Any] = <factory>*, *enabled: bool = True*)

Bases: `object`

Database section content mirrored from the steering file.

Parameters

- **url** – Optional database URL connection string.
- **pragmas** – SQLite pragma settings to apply on connection.
- **authentication** – Optional authentication subtable for server-based backends.
- **parameters** – Optional per-backend configuration parameters.
- **attributes** – Additional arbitrary attributes for the database configuration.
- **enabled** – Whether `DBConfiguration` should be serialized.

class mafw.steering.models.**FieldFilterConfig**(*name: str*, *enabled: bool = True*, *model: str = ''*, *field_name: str = ''*, *conditions: dict[str, Condition] = <factory>*, *logic_str_original: str | None = None*, *logic_buffer: str | None = None*, *logic_ast: ExprNode | None = None*, *logic_dirty: bool = False*, *logic_is_valid: bool = True*)

Bases: *FilterConfig*

Filter applied to a single field of a specific model.

Conditions are AND'ed by default unless a custom logical expression is defined via *logic*.

property logic: str | None

Backward compatibility for logic string.

class mafw.steering.models.**FieldInfo**(*name: str*, *type: type*, *help_text: str | None = None*)

Bases: object

Metadata about a model field.

class mafw.steering.models.**FilterConfig**(*name: str*, *enabled: bool = True*, *model: str = ''*)

Bases: ABC

Base configuration for a filter.

class mafw.steering.models.**FilterKind**(**values*)

Bases: StrEnum

The kind of filter configuration.

static _generate_next_value_(*name*, *start*, *count*, *last_values*)

Return the lower-cased version of the member name.

class mafw.steering.models.**GlobalSettings**(*processors_to_run: list[str] = <factory>*, *analysis_name: str | None = None*, *analysis_description: str | None = None*, *new_only: bool | None = None*, *create_standard_tables: bool | None = None*)

Bases: object

Top-level globals that appear outside any processor or group table.

Parameters

- **processors_to_run** – Ordered list of processor or group names to execute.
- **analysis_name** – Optional name for the analysis.
- **analysis_description** – Optional description for the analysis.
- **new_only** – Optional flag to process only new data.
- **create_standard_tables** – Optional flag to create standard output tables.

class mafw.steering.models.**GroupConfig**(*name: str, processors: list[str], description: str | None = None, attributes: dict[str, ~typing.Any] = <factory>*)

Bases: object

Configuration for a logical processor group section.

Parameters

- **name** – Section name used to reference the group in processors_to_run.
- **processors** – Ordered list of processor names and/or nested group names.
- **description** – Optional documentation string for the group.
- **attributes** – Additional arbitrary attributes for the group.

class mafw.steering.models.**ModelFilterConfig**(*name: str*, *enabled: bool = True*, *model: str = ''*, *conditions: dict[str, Condition] = <factory>*, *logic_str_original: str | None = None*, *logic_buffer: str | None = None*, *logic_ast: ExprNode | None = None*, *logic_dirty: bool = False*, *logic_is_valid: bool = True*)

Bases: *FilterConfig*

Filter applied to a specific database model.

Each field condition is AND'ed by default unless a custom logical expression is provided via *logic*.

fill_model_info() → None

Populate model_info by inspecting the database model.

property logic: str | None

Backward compatibility for logic string.

class mafw.steering.models.**ModelInfo**(*field_info: dict[str, ~mafw.steering.models.FieldInfo] = <factory>*)

Bases: object

Metadata about a database model.

class mafw.steering.models.**ParameterConfig**(*name: str*, *value: Any = None*, *default: Any = None*, *source:* ParameterSource *= ParameterSource.CONFIG*, *status:* ParameterSchemaStatus *= ParameterSchemaStatus.OK*, *help: str | None = None*, *active_override: bool = False*, *type: Any = None*)

Bases: object

Configuration and status for a single processor parameter.

Parameters

- **name** – The name of the parameter.
- **value** – The current value of the parameter.
- **default** – The default value of the parameter.
- **source** – The source of the parameter value.
- **status** – The schema reconciliation status of the parameter.
- **help** – Documentation text for the parameter.
- **type** – The type hint for the parameter.

class mafw.steering.models.**ParameterSchemaStatus**(**values*)

Bases: StrEnum

Schema reconciliation status for a processor parameter.

static _generate_next_value_(*name*, *start*, *count*, *last_values*)

Return the lower-cased version of the member name.

class mafw.steering.models.**ParameterSource**(**values*)

Bases: StrEnum

Origin of a processor parameter value in the GUI pipeline.

static _generate_next_value_(*name*, *start*, *count*, *last_values*)

Return the lower-cased version of the member name.

class mafw.steering.models.**ProcessorConfig**(*name: str*, *parameters: dict[str, ParameterConfig] = <factory>*, *processor_status: ProcessorSchemaStatus = ProcessorSchemaStatus.OK*, *filters: dict[str, list[FilterConfig]] = <factory>*, *logic_str_original: str | None = None*, *logic_buffer: str | None = None*, *logic_ast: ExprNode | None = None*, *logic_dirty: bool = False*, *logic_is_valid: bool = True*, *new_only: bool | None = None*, *inheritance: bool | None = None*, *has_filter_root: bool = True*, *schema_filter_models: list[str] = <factory>*)

Bases: object

Configuration store for an individual processor or replica section.

Parameters

- **name** – Replica-aware section name (`Processor` or `Processor#Replica`).
- **parameters** – Parameter configurations and status.
- **processor_status** – Schema reconciliation status for the processor.
- **filters** – Mapping from model name to list of filter configuration objects.
- **logic_str_original** – Optional `__logic__` string stored in the per-filter table.
- **new_only** – Optional `__new_only__` override.
- **inheritance** – Optional `__inheritance__` flag for replicas.
- **has_filter_root** – Whether the filter table includes a root entry.

property filter_logic: str | None

Backward compatibility for filter_logic string.

class mafw.steering.models.**ProcessorRef**(*base_name: str*, *replica: str | None = None*)

Bases: `object`

Reference to a processor entry used in processors_to_run lists.

Parameters

- **base_name** – The canonical name of the processor without replica suffix.
- **replica** – Optional replica identifier appended to the base name with `#`.

property full_name: str

Return the replica-aware full identifier for this reference.

class mafw.steering.models.**ProcessorSchemaStatus**(**values*)

Bases: `StrEnum`

Schema reconciliation status for a processor.

static _generate_next_value_(*name*, *start*, *count*, *last_values*)

Return the lower-cased version of the member name.

class mafw.steering.models.**UIConfiguration**(*interface: str = 'rich'*)

Bases: `object`

User-interface section content mirrored from the steering file.

Parameters

interface – The interface type to use (e.g., `rich`).

mafw.steering.models.**_field_type_from_peewee**(*field_obj: Any*) → type

Extract a primitive type from the given Peewee field definition.

mafw.steering.models.**_normalize_field_type**(*field_type: Any*) → type

Normalize a field descriptor to a supported Python primitive.

mafw.steering.models._FIELD_TYPE_STRINGS: dict[str, type] = {'AUTO': <class 'int'>, 'BIGINT': <class 'int'>, 'BOOL': <class 'bool'>, 'BOOLEAN': <class 'bool'>, 'CHAR': <class 'str'>, 'DATE': <class 'str'>, 'DATETIME': <class 'str'>, 'DECIMAL': <class 'float'>, 'DOUBLE': <class 'float'>, 'FLOAT': <class 'float'>, 'INT': <class 'int'>, 'INTEGER': <class 'int'>, 'NUMERIC': <class 'float'>, 'REAL': <class 'float'>, 'SMALLINT': <class 'int'>, 'STRING': <class 'str'>, 'TEXT': <class 'str'>, 'TIME': <class 'str'>, 'TINYINT': <class 'int'>, 'VARCHAR': <class 'str'>}

Mapping from Peewee field_type strings to Python primitives.

mafw.steering.serializer

Serialize steering metadata into TOML documents via tomlkit.

Functions

serialize(builder, *[, validation_level])	Convert builder metadata into a TOML document matching the required layout.

mafw.steering.serializer.**_normalize_logical_op**(*value: str* | LogicalOp) → *LogicalOp* | None

Return the LogicalOp enum for the provided value when possible.

mafw.steering.serializer.**_serialize_condition**(*condition:* Condition) → Any

Convert Condition object to TOML-friendly value (implicit or dict).

mafw.steering.serializer.**serialize**(*builder:* SteeringBuilder, *, *validation_level:* ValidationLevel | *None* = *None*) → tomlkit.TOMLDocument

Convert builder metadata into a TOML document matching the required layout.

mafw.steering.validation

Structural validation helpers for steering metadata.

Functions

validate(builder[, validation_level])	Run steering validation and return every issue found.

mafw.steering.validation.**validate**(*builder:* SteeringBuilder, *validation_level:* ValidationLevel = *ValidationLevel.SEMANTIC*) → list[*ValidationIssue*]

Run steering validation and return every issue found.

14.1.18 mafw.steering_gui

Application layer for steering GUI orchestration.

Author

Bulgheroni Antonio (antonio.bulgheroni@ec.europa.eu)

Description

Expose the steering GUI controller without leaking UI widgets or execution logic.

Modules

actions	Utility helpers for creating the steering GUI actions without duplicating wiring.
app	Entry point for the steering GUI application.
controllers	Container for steering GUI application controllers.
dialogs	Dialog helpers for the steering GUI package.
main_window	Main window orchestrator for the steering GUI layout.
models	GUI-facing models and Qt wrappers for steering metadata.
threading	Threading utilities made available to the steering GUI.
utils	Helper utilities for the steering GUI.
views	Placeholder package for steering GUI views while the GUI is being scaffolded.

mafw.steering_gui.actions

Utility helpers for creating the steering GUI actions without duplicating wiring.

Author
Bulgheroni Antonio (antonio.bulgheroni@ec.europa.eu)

Description
Share QAction definitions so the menu wiring stays clean.

Functions

create_action(parent, spec)	Instantiate a QAction from the provided *ActionSpec*.

Classes

ActionSpec(text, triggered[, shortcut, ...])	Describe the metadata needed to build a menu action.

class mafw.steering_gui.actions.**ActionSpec**(*text: str*, *triggered: Callable[[], None]*, *shortcut: StandardKey | str | None = None*, *status_tip: str | None = None*, *enabled: bool = True*)

Bases: `object`

Describe the metadata needed to build a menu action.

enabled: bool = True
Initial enabled state for the action.

shortcut: StandardKey | str | None = None
Optional `QKeySequence.StandardKey` or str describing the shortcut.

status_tip: str | None = None
Optional text for the status bar.

text: str
The label shown in the menu.

triggered: Callable[[], None]
Callback invoked when the user activates the action.

mafw.steering_gui.actions.**create_action**(*parent: QObject*, *spec:* ActionSpec) → QAction
Instantiate a QAction from the provided *ActionSpec*.

mafw.steering_gui.app

Entry point for the steering GUI application.

Author
Bulgheroni Antonio (antonio.bulgheroni@ec.europa.eu)

Description
Verify dependencies, start QApplication, and show MainWindow.

Functions

main()	Entry point that creates the QApplication and displays the main window.

mafw.steering_gui.app.**main**() → None
Entry point that creates the QApplication and displays the main window.

mafw.steering_gui.controllers

Container for steering GUI application controllers.

Author
Bulgheroni Antonio (antonio.bulgheroni@ec.europa.eu)

Description
Hosts orchestrators that expose domain-safe operations to the UI layer.

Modules

steering_controller	Application layer controller for steering configuration builds.

mafw.steering_gui.controllers.steering_controller

Application layer controller for steering configuration builds.

Author
Bulgheroni Antonio (antonio.bulgheroni@ec.europa.eu)

Description
Keep GUI and execution logic isolated while managing SteeringBuilder state.

Module Attributes

P	Type variable capturing the positional parameters of the wrapped method.
R	Type variable capturing the return type of the wrapped method.

Functions

build_database_connection_configuration(co	Return a normalized configuration dict for connection checks.
recorded_mutation(method)	Decorator that records the controller state before mutating operations.

Classes

DirtyStateSignal()	Lightweight notifier for controller dirty-state transitions.
SteeringController([builder, max_history, ...])	Controller orchestrating SteeringBuilder without GUI or execution ties.

Exceptions

SteeringControllerError	Raised when the controller is asked to perform an invalid operation.

exception mafw.steering_gui.controllers.steering_controller.**SteeringControllerError**

Bases: *MAFwException*

Raised when the controller is asked to perform an invalid operation.

class mafw.steering_gui.controllers.steering_controller.**DirtyStateSignal**

Bases: object

Lightweight notifier for controller dirty-state transitions.

connect(*callback: Callable[[bool], None]*) → None

Subscribe to dirty-state change notifications.

emit(*value: bool*) → None

Notify every subscriber about the new dirty value.

class mafw.steering_gui.controllers.steering_controller.**R**

Type variable capturing the return type of the wrapped method.

alias of TypeVar('R')

class mafw.steering_gui.controllers.steering_controller.**SteeringController**(*builder: SteeringBuilder | None = None*, ***, *max_history: int | None = None*, *plugin_manager: MAFwPluginManager | None = None*)

Bases: object

Controller orchestrating SteeringBuilder without GUI or execution ties.

classmethod from_file(*path: Path | str*) → *SteeringController*

Instantiate a controller from an existing steering file.

static _normalize_schema_default(*value: Any*) → Any

Convert schema defaults into serializable values.

_assemble_connection_source(*overrides: dict[str, Any] | None = None*) → dict[str, Any]

Combine controller-managed attributes with override input.

_mark_clean() → None

Mark the controller as clean and notify subscribers.

_unique_processor_name(*base_name: str*) → tuple[str, str | None]

Return a unique processor name and replica identifier for the requested base.

_write_to_path(*path: Path*) → None

Serialize the builder to the provided path.

add_group(*name: str*, *processors: Iterable[str]*, *description: str | None = None*) → None

Declare a new group section.

add_group_entry_to_group(*target_group: str*, *entry_name: str*) → None

Append a processor or group reference to a group list.

add_group_to_run_list(*name: str*) → None

Append a group name to the processors_to_run list if missing.

add_processor_to_run_list(*base_name: str*, *replica: str | None = None*) → None

Add a processor or replica to the execution list and ensure its section exists.

add_processors_to_group_with_defaults(*group_name: str*, *base_names: Iterable[str]*) → None

Append processors to a group and set their parameters to schema defaults.

add_processors_with_defaults(*base_names: Iterable[str]*) → None

Append processors to the run list and set their parameters to schema defaults.

add_replica(*base_name: str*, *replica: str*) → None

Ensure a replica section exists without altering parameters.

available_db_module_count() → int | None

Return how many database modules have been loaded (or `None` until available).

available_processor_count() → int | None

Return how many processor plugins have been loaded (or `None` until available).

available_processor_names() → list[str]

Return the available processor names from loaded plugins.

available_ui_count() → int | None

Return how many user interface plugins have been loaded (or `None` until available).

build_connection_test_task(*config: dict[str, Any] | None = None*) → Callable[[], bool]

Return a callable that validates the current database settings via Peewee.

Parameters

config – Optional overrides supplied by the GUI connection tester.

Returns

Callable that returns True when the connection is verified; it raises on failure.

build_pipeline(*, *progress_callback: Callable[[int], None] | None = None*) → *ProcessorPipeline*

Build the in-memory pipeline tree for the steering GUI.

can_redo() → bool

Return whether a previously undone state can be re-applied.

can_undo() → bool

Return whether the controller can revert to a previously recorded state.

clear_processor_parameters(*full_name: str*) → None

Clear every parameter override for a processor.

current_file() → Path | None

Return the file currently associated with persistence operations.

disable_db_configuration() → None

Disable the DBConfiguration section so it is omitted when written.

enable_db_configuration() → None

Enable the DBConfiguration section for serialization.

ensure_valid(*validation_level:* ValidationLevel = *ValidationLevel.SEMANTIC*) → None

Raise the first validation issue or return silently when the builder is valid.

export_to_path(*path: Path | str*) → None

Serialize the builder to the provided path without validation.

get_analysis_description() → str | None

Return the configured analysis_description safely.

get_analysis_name() → str | None

Return the configured analysis_name without exposing the builder.

get_create_standard_tables() → bool | None

Return the create_standard_tables flag.

get_db_attributes() → dict[str, Any]

Return a copy of the database attributes map.

get_db_authentication() → dict[str, Any]

Return a copy of the DB authentication mapping.

get_db_parameters() → dict[str, dict[str, Any]]

Return a copy of the DB backend parameters mapping.

get_db_pragmas() → dict[str, Any]

Return a copy of the DB pragmas map.

get_db_url() → str | None

Return the configured database URL.

get_globals() → dict[str, Any]

Return the globals metadata used by the UI.

get_group_attributes(*name: str*) → dict[str, Any]

Return a copy of a group's attributes.

get_group_snapshot(*name: str*) → *GroupConfig*

Return a copy of a group configuration.

get_new_only() → bool | None

Return the current new_only flag.

get_processor_snapshot(*full_name: str*) → *ProcessorConfig*

Return a deep copy snapshot of a processor configuration.

get_processors_to_run() → list[str]

Return a copy of the ordered processor references.

get_ui_interface() → str

Return the configured UI interface identifier.

is_db_configuration_enabled() → bool

Return whether the DBConfiguration section will appear in serialized output.

list_groups() → list[str]

Return the declared group section names.

list_processors() → list[str]

Return the known processor section names.

load(*path: Path | str*) → None

Load steering metadata from disk, replace the current builder, and rebase the reset snapshot.

load_plugins() → *LoadedPlugins*

Load the asynchronous plugin bundles and cache the result for GUI consumers.

Returns

Plugins container with available processors, database modules, and UIs

Return type

LoadedPlugins

move_pipeline_entry(*entry_name: str*, *entry_is_group: bool*, *source_group: str | None*, *target_group: str | None*, *position: int*) → None

Move a pipeline entry across parents and adjust execution ordering.

move_processor_in_run_list(*full_name: str*, *position: int*) → None

Move a scheduled processor to a concrete slot in the execution order.

redo() → None

Re-apply the last undone snapshot.

remove_filter(*full_name: str*, *model_name: str*) → None

Delete a filter model definition.

remove_group(*name: str*) → None

Delete a named group section.

remove_pipeline_entries(*names: Iterable[str]*) → None

Remove pipeline entries from the top-level processors_to_run list.

remove_pipeline_entry(*entry_name: str*, *source_group: str | None*) → None

Remove a pipeline entry from the top-level or a group list.

remove_processor_from_run_list(*full_name: str*) → None

Remove a processor reference from the execution order.

remove_processor_parameter(*full_name: str*, *key: str*) → None

Remove a processor parameter override if present.

rename_group(*old_name: str*, *new_name: str*) → None

Rename a group and update all pipeline references.

rename_processor_entry(*old_name: str*, *new_name: str*, *source_group: str | None*) → None

Rename a processor reference and keep its configuration aligned.

replace_builder(*builder:* SteeringBuilder) → None

Replace the current builder with the provided instance.

reset() → None

Restore the controller to its initial state and clear history.

reset_processor_parameters(*full_name: str*) → None

Reset the processor parameters to their default values without dropping the overrides.

save(*, *skip_validation: bool = False*) → None

Persist the current steering state to the associated file path.

save_as(*path: Path | str*, *, *skip_validation: bool = False*) → None

Persist the current steering state to a new file path and track it.

set_analysis_description(*value: str | None*) → None

Update the top-level analysis_description setting.

set_analysis_name(*value: str | None*) → None

Update the top-level analysis_name setting.

set_create_standard_tables(*value: bool | None*) → None

Update the create_standard_tables flag.

set_db_attribute(*key: str*, *value: Any | None*) → None

Update a single entry stored under DBConfiguration.

set_db_authentication(*authentication: Mapping[str, Any] | None*) → None

Replace the DB authentication mapping with the provided data.

set_db_parameters(*parameters: Mapping[str, Mapping[str, Any]] | None*) → None

Replace the DB parameters mapping with the provided data.

set_db_pragmas(*pragmas: Mapping[str, Any]*) → None

Replace the DB pragmas map with a copy of the provided mapping.

set_db_url(*value: str | None*) → None

Update the DB URL while keeping builder semantics.

set_filter_conditionals(*full_name: str*, *model_name: str*, *conditionals: list[Mapping[str, Any]] | None*) → None

Assign __conditional__ blocks to a filter model.

set_filter_config(*full_name: str*, *model_name: str*, *config: Mapping[str, Any]*) → None

Replace a filter model configuration with a copy of the provided mapping.

set_filter_field(*full_name: str*, *model_name: str*, *field: str*, *value: Any*) → None

Set or update a single filter field.

set_filter_logic(*full_name: str*, *logic: str | None*) → None

Update the __logic__ string for a processor's filters.

set_group_attributes(*name: str*, *attributes: Mapping[str, Any]*) → None

Replace a group's attributes with a new copy.

set_group_description(*name: str*, *description: str | None*) → None

Set or clear the description for a group.

set_group_processors(*name: str*, *processors: Iterable[str]*) → None

Replace the processors list for the given group.

set_new_only(*value: bool | None*) → None

Update the new_only flag.

set_processor_filters(*full_name: str*, *filters: Mapping[str, Iterable[FilterConfig]]*, *logic: str | None*) → None

Replace the whole filter dictionary and logic for a processor.

set_processor_new_only(*full_name: str*, *value: bool | None*) → None

Set the __new_only__ override for a processor section.

set_processor_parameter(*full_name: str*, *key: str*, *value: Any*) → None

Update a single processor parameter.

set_processors_to_run(*processors: Iterable[str]*) → None

Overwrite the processors_to_run list while keeping a copy of the iterable.

set_replica_inheritance(*replica_full_name: str*, *value: bool | None*) → None

Toggle replica inheritance.

set_ui_interface(*interface: str*) → None

Update the selected UI interface (neutral placeholder, no GUI imports).

set_validation_level(*level: ValidationLevel | None*) → None

Keep track of the validation level preference without affecting builder data.

undo() → None

Revert the builder to the previous snapshot.

validate(*validation_level: ValidationLevel = ValidationLevel.SEMANTIC*) → list[*ValidationIssue*]

Run validation and return every detected issue.

validation_level() → *ValidationLevel* | None

Return the validation level currently configured on the controller.

property dirty: bool

Expose whether the controller currently tracks unsaved changes.

mafw.steering_gui.controllers.steering_controller.**build_database_connection_configuration**(*config: Mapping[str, Any]*) → dict[str, Any]

Return a normalized configuration dict for connection checks.

The returned dictionary always contains `URL` plus backend-specific keys derived from the configuration schema.

mafw.steering_gui.controllers.steering_controller.**recorded_mutation**(*method: Callable[[Concatenate[SteeringController, P]], R]*) → Callable[[Concatenate[*SteeringController*, *P*]], *R*]

Decorator that records the controller state before mutating operations.

mafw.steering_gui.controllers.steering_controller.**P** = ~P

Type variable capturing the positional parameters of the wrapped method.

mafw.steering_gui.dialogs

Dialog helpers for the steering GUI package.

Author
Bulgheroni Antonio (antonio.bulgheroni@ec.europa.eu)

Description
expose dialog widgets such as AboutDialog to other modules.

class mafw.steering_gui.dialogs.**AboutDialog**(*parent: QWidget | None = None*)

Bases: `QDialog`

Show the About page for MAFw and the steering GUI.

Parameters
parent – Optional parent widget for window hierarchy.

_setup_ui() → None

Compose the dialog layout and populate the informative labels.

class mafw.steering_gui.dialogs.**AddProcessorDialog**(*processors: list[str]*, *parent: QWidget | None = None*)

Bases: `QDialog`

Dialog presenting a selectable list of processors.

selected_names() → list[str]

Return the currently selected processor names.

class mafw.steering_gui.dialogs.**SteeringTextEditor**(*controller: SteeringController*, *parent: QWidget | None = None*)

Bases: `QDialog`

Dialog that lets users edit the steering file directly as TOML.

Modules

about_dialog	Dialog presenting metadata about the Steering GUI application.
add_field_filter_dialog	Dialog for selecting fields for field filters.
add_model_dialog	Dialog for selecting filter models to add.
add_processor_dialog	Dialog for selecting processors to add to the pipeline.
steering_text_editor	Dialog for manual editing of steering files in TOML format.
validation_issues_dialog	Dialog that lets the user review validation issues and optionally save anyway.

mafw.steering_gui.dialogs.about_dialog

Dialog presenting metadata about the Steering GUI application.

Author
Bulgheroni Antonio (antonio.bulgheroni@ec.europa.eu)

Description
Display icon, project description, URLs, and authorship to the user.

Classes

AboutDialog([parent])	Show the About page for MAFw and the steering GUI.

class mafw.steering_gui.dialogs.about_dialog.**AboutDialog**(*parent: QWidget | None = None*)

Bases: QDialog

Show the About page for MAFw and the steering GUI.

Parameters
parent – Optional parent widget for window hierarchy.

_setup_ui() → None

Compose the dialog layout and populate the informative labels.

mafw.steering_gui.dialogs.add_field_filter_dialog

Dialog for selecting fields for field filters.

Classes

AddFieldFilterDialog(model_config, ...[, parent])	Dialog presenting a selectable list of fields for a model.

class mafw.steering_gui.dialogs.add_field_filter_dialog.**AddFieldFilterDialog**(*model_config: ModelFilterConfig, processor_config: ProcessorConfig, parent: QWidget | None = None*)

Bases: QDialog

Dialog presenting a selectable list of fields for a model.

selected_field() → str | None

Return the currently selected field name.

mafw.steering_gui.dialogs.add_model_dialog

Dialog for selecting filter models to add.

Classes

AddModelDialog(advertised_models, all_models)	Dialog presenting a selectable list of models.

class mafw.steering_gui.dialogs.add_model_dialog.**AddModelDialog**(*advertised_models: list[str]*, *all_models: list[str]*, *parent: QWidget | None = None*)

Bases: QDialog

Dialog presenting a selectable list of models.

selected_model() → str | None

Return the currently selected model name.

mafw.steering_gui.dialogs.add_processor_dialog

Dialog for selecting processors to add to the pipeline.

Author

Bulgheroni Antonio (antonio.bulgheroni@ec.europa.eu)

Description

Display available processors in a list and return the selected names.

Classes

AddProcessorDialog(processors[, parent])	Dialog presenting a selectable list of processors.

class mafw.steering_gui.dialogs.add_processor_dialog.**AddProcessorDialog**(*processors: list[str]*, *parent: QWidget | None = None*)

Bases: QDialog

Dialog presenting a selectable list of processors.

selected_names() → list[str]

Return the currently selected processor names.

mafw.steering_gui.dialogs.steering_text_editor

Dialog for manual editing of steering files in TOML format.

Author

Bulgheroni Antonio (antonio.bulgheroni@ec.europa.eu)

Description

Allow advanced users to edit the TOML serialization of the current steering builder.

Classes

SteeringTextEditor(controller[, parent])	Dialog that lets users edit the steering file directly as TOML.
TomlSyntaxHighlighter([parent])	Simple TOML syntax highlighter for the manual editor.

class mafw.steering_gui.dialogs.steering_text_editor.**SteeringTextEditor**(*controller: SteeringController*, *parent: QWidget | None = None*)

Bases: QDialog

Dialog that lets users edit the steering file directly as TOML.

class mafw.steering_gui.dialogs.steering_text_editor.**TomlSyntaxHighlighter**(*parent: Any = None*)

Bases: QSyntaxHighlighter

Simple TOML syntax highlighter for the manual editor.

mafw.steering_gui.dialogs.steering_text_editor.**_VALIDATION_DEBOUNCE_MS = 500**

Debounce duration for TOML validation in milliseconds.

mafw.steering_gui.dialogs.validation_issues_dialog

Dialog that lets the user review validation issues and optionally save anyway.

Classes

ValidationIssuesDialog(issues[, parent])	Present every validation issue and allow the user to cancel or save anyway.

class mafw.steering_gui.dialogs.validation_issues_dialog.**ValidationIssuesDialog**(*issues: Sequence[ValidationIssue]*, *parent: QWidget | None = None*)

Bases: QDialog

Present every validation issue and allow the user to cancel or save anyway.

_adjust_column_widths() → None

Resize each column once so every issue text fits.

mafw.steering_gui.main_window

Main window orchestrator for the steering GUI layout.

Author

Bulgheroni Antonio (antonio.bulgheroni@ec.europa.eu)

Description

Build the menu, splitter, tree, stacked editors, and status bar required by the GUI.

Classes

MainWindow([controller, parent])	Compose the steering GUI main window without executing controller logic.

class mafw.steering_gui.main_window.**MainWindow**(*controller: SteeringController | None = None*, *parent: QWidget | None = None*)

Bases: QMainWindow

Compose the steering GUI main window without executing controller logic.

static _normalize_text(*value: str*) → str | None

Treat empty or whitespace-only strings as unset.

_confirm_discard() → bool

Ask whether the user wants to save/discard/cancel when the document is dirty.

_create_actions() → None

Declare the main window actions and store them for later menu wiring.

_create_central_widget() → None

Build the splitter containing the tree and editor stack.

_create_group(*parent_group: str | None*) → str | None

Create a new group and attach it to the requested parent list.

_create_menus() → None

Compose the menu bar using the actions previously created.

_create_status_bar() → None

Initialise the status bar to keep the user informed.

_display_section(*section: str*) → None

Show the editor that corresponds to the selected tree section.

_exit_app() → None

Close the window after ensuring dirty state obeys the user.

_finalize_connection_test(*result: object | None = None*) → None

Reset worker references and re-enable the test button.

_generate_group_name() → str

Return a unique group name based on the current builder state.

_load_file(*path: Path*) → None

Load the steering file through the controller and refresh the UI.

_new_file() → None

Reset the UI state in preparation for a new steering file.

_on_add_group_requested() → None

Create a new top-level group and open it in the editor.

_on_add_processor_requested() → None

Open the Add Processor dialog and append selections to the pipeline.

_on_connection_test_failure(*exc: Exception*) → None

Report a failure with the provided exception message.

_on_connection_test_success(*_: object*) → None

Report a successful connection and reset the UI state.

_on_database_configuration_changed(*data: dict[str, Any]*) → None

Commit every database widget change back to the controller.

_on_dirty_changed(*dirty: bool*) → None

Update the window title when the controller dirty state changes.

_on_group_add_group_requested() → None

Create a new group nested within the currently edited group.

_on_group_add_processor_requested() → None

Add processors to the currently edited group.

_on_group_description_changed(*description: str*) → None

Update the description for the currently edited group.

_on_group_move_requested(*direction: str*, *items: list[*PipelineItem*]*) → None

Reorder items inside the currently edited group.

_on_group_name_changed(*new_name: str*) → None

Rename the currently edited group.

_on_group_remove_requested(*items: list[*PipelineItem*]*) → None

Remove items from the currently edited group.

_on_pipeline_edit_requested(*item:* PipelineItem) → None

Switch to the appropriate placeholder editor for the selected item.

_on_pipeline_error(*exc: Exception*) → None

Report pipeline building failures to the user.

_on_pipeline_move_requested(*direction: str*, *items: list[*PipelineItem*]*) → None

Reorder the selected items within the top-level pipeline list.

_on_pipeline_ready(*pipeline:* ProcessorPipeline) → None

Update the tree and editors with the built pipeline.

_on_pipeline_remove_requested(*items: list[*PipelineItem*]*) → None

Remove selected items from the top-level processors_to_run list.

_on_pipeline_selection_changed(*items: list[*PipelineItem*]*) → None

Switch editors when a pipeline item is selected in the tree.

_on_plugin_load_failure(*exc: Exception*) → None

Report plugin loading failures to the user and refresh the UI.

_on_plugins_loaded(*_: object*) → None

Handle successful plugin loading and refresh the UI.

_on_test_connection_requested(*config: dict[str, Any]*) → None

Start an asynchronous database connection test and update the UI.

_on_ui_interface_changed(*interface: str*) → None

Tell the controller to use the selected UI interface.

_on_validation_level_changed(*level:* ValidationLevel | *None*) → None

Tell the controller about the user-selected validation level.

_open_file() → None

Stub that requests a filename without yet loading it.

_open_steering_text_editor() → None

Open the manual TOML editor for advanced steering edits.

_perform_save() → bool
Persist the current builder to its tracked path (or prompt Save As).

_perform_save_as() → bool
Prompt the user for a location and persist the current builder there.

_populate_from_model() → None
Refresh the tree and metadata editor from the controller model.

_redo() → None
Placeholder redo action until controller wiring occurs.

_refresh_database_editor() → None
Push the controller database metadata into the database editor.

_refresh_globals_editor() → None
Push the controller globals data into the globals editor.

_refresh_metadata_summary() → None
Update the metadata view with configured and available plugin data.

_refresh_pipeline_after_change(*select_name: str | None = None*) → None
Rebuild the pipeline and refresh dependent widgets.

_refresh_title_from_controller(*dirty: bool | None = None*) → None
Compose the window title with the associated file and dirty marker.

_refresh_ui_editor() → None
Update the UI combobox once plugin metadata or configuration changes.

_restore_window_geometry() → None
Restore the last-window size and position when available.

_save_file() → None
Action handler that keeps the QAction signature while delegating to DoSave.

_save_file_as() → None
Action handler for Save As that satisfies the QAction signature.

_save_window_geometry() → None
Persist the current geometry before exit.

_set_test_button_enabled(*enabled: bool*) → None
Toggle the database editor test button when the feature is active.

_show_about() → None
Display the About dialog describing the GUI.

_start_connection_test_worker(*task: Callable[[], bool]*) → None
Start the worker via a thread pool and bind completion callbacks.

_start_plugin_loading() → None
Trigger the controller to load plugins in the background.

_undo() → None
Placeholder undo action until controller wiring occurs.

_wire_controller_signals() → None
Wire controller dirty notifications and globals signals to the editor.

closeEvent(*event: QCloseEvent*) → None
Prompt the user to save if the current state is dirty.

mafw.steering_gui.main_window.**_DEFAULT_SAVE_NAME = 'mysteerfile.toml'**
Default suggestion when the user invokes Save As.

mafw.steering_gui.models

GUI-facing models and Qt wrappers for steering metadata.

Author
Bulgheroni Antonio

Description
Holds the editable UI models that wrap steering builder domain objects.

class mafw.steering_gui.models.**PipelineItem**(*config: ~mafw.steering.models.ProcessorConfig | ~mafw.steering.models.GroupConfig*, *parent: ~mafw.steering_gui.models.pipeline.PipelineItem | None = None*, *children: list[~mafw.steering_gui.models.pipeline.PipelineItem] = <factory>*)

Bases: `object`

UI wrapper around a processor or group configuration.

Parameters

- **config** – Domain configuration backing the item.
- **parent** – Parent pipeline item (None when top-level).
- **children** – Ordered list of child items.

is_group() → bool

Return True if this item wraps a group configuration.

name() → str

Return the replica-aware processor name or group name.

class mafw.steering_gui.models.**ProcessorPipeline**(*items: list[~mafw.steering_gui.models.pipeline.PipelineItem] = <factory>*)

Bases: `object`

Container for the current processor pipeline tree.

Parameters
items – Top-level pipeline items in execution order.

class mafw.steering_gui.models.**ProcessorPipelineModel**(*pipeline: ProcessorPipeline | None = None*, *parent: QObject | None = None*)

Bases: `QAbstractItemModel`

Expose a ProcessorPipeline as a tree-structured Qt model.

pipeline() → *ProcessorPipeline*

Return the currently attached pipeline.

root_items() → list[*PipelineItem*]

Return the top-level pipeline items.

set_pipeline(*pipeline: ProcessorPipeline*) → None

Replace the pipeline data and reset the model.

class mafw.steering_gui.models.**SteeringTreeModel**(*controller: object*, *pipeline: ProcessorPipeline | None = None*, *parent: QObject | None = None*)

Bases: `QAbstractItemModel`

Tree model exposing the steering sections and embedded processor pipeline.

pipeline_item_from_index(*index: QModelIndex | QPersistentModelIndex*) → *PipelineItem* | None

Return the pipeline item stored at the given index, if any.

section_key_from_index(*index: QModelIndex | QPersistentModelIndex*) → str | None

Return the section key stored at the given index, if any.

set_pipeline(*pipeline:* ProcessorPipeline) → None

Replace the pipeline subtree and reset the model.

class mafw.steering_gui.models.**SteeringTreeRoles**(**values*)

Bases: IntEnum

Custom roles exported by the steering tree model.

Modules

filter_tree_model	Tree model representing the filter configuration hierarchy.
pipeline	UI-only pipeline structures used to render editable processor hierarchies.
processor_pipeline_model	Qt model for the processor pipeline tree.
steering_tree_model	Composite Qt model that embeds the processor pipeline under the steering tree.

mafw.steering_gui.models.filter_tree_model

Tree model representing the filter configuration hierarchy.

Author
Bulgheroni Antonio (antonio.bulgheroni@ec.europa.eu)

Description
QAbstractItemModel implementation for displaying and editing filters, conditions, and logic associated with a processor.

Classes

ConditionKey(model_name, field_name, ...)	Stable key describing a condition entry in the filter tree.
FilterNodeType(*values)	Types of nodes in the filter tree.
FilterTreeItem(node_type[, data, parent, key])	Internal node for the FilterTreeModel.
FilterTreeModel(config[, parent])	Tree model for processor filters.
LogicKey(model_name, field_name)	Stable key describing a logic entry in the filter tree.

class mafw.steering_gui.models.filter_tree_model.**ConditionKey**(*model_name: str*, *field_name: str | None*, *condition_name: str*)

Bases: object

Stable key describing a condition entry in the filter tree.

class mafw.steering_gui.models.filter_tree_model.**FilterNodeType**(**values*)

Bases: Enum

Types of nodes in the filter tree.

class mafw.steering_gui.models.filter_tree_model.**FilterTreeItem**(*node_type: FilterNodeType*, *data: Any = None*, *parent: FilterTreeItem | None = None*, *key: str | None = None*)

Bases: `object`

Internal node for the FilterTreeModel.

class mafw.steering_gui.models.filter_tree_model.**FilterTreeModel**(*config: ProcessorConfig*, *parent: Any = None*)

Bases: `QAbstractItemModel`

Tree model for processor filters.

_collect_condition_keys() → set[*ConditionKey*]

Return the set of condition keys currently in the tree.

_collect_logic_keys() → set[*LogicKey*]

Return the set of logic keys currently in the tree.

_has_logic(*config: ProcessorConfig | ModelFilterConfig | FieldFilterConfig*) → bool

Return whether the config currently has a logic buffer or original string.

_setup_model_data() → None

Build the tree structure from the ProcessorConfig.

collect_subtree_indexes(*parent_index: QModelIndex | QPersistentModelIndex*) → list[QModelIndex | QPersistentModelIndex]

Return every index below the provided parent.

condition_key_for_index(*index: QModelIndex | QPersistentModelIndex*) → *ConditionKey* | None

Return a condition key for the provided index.

condition_key_for_item(*item: FilterTreeItem*) → *ConditionKey* | None

Return a condition key for the provided tree item.

condition_key_for_parent(*parent: ModelFilterConfig | FieldFilterConfig*, *condition_name: str*) → *ConditionKey* | None

Return the condition key for the given parent config and name.

find_condition_index(*key: ConditionKey*) → QModelIndex

Return the model index for a condition key if present.

find_index_for_data(*data: ~typing.Any*, *parent_index: ~PySide6.QtCore.QModelIndex = <PySide6.QtCore.QModelIndex(-1, -1, 0x0, QObject(0x0))>*) → QModelIndex

Find the QModelIndex for the given data object by searching the tree recursively.

find_logic_index(*key: LogicKey*) → QModelIndex

Return the model index for a logic key if present.

find_model_index(*model_name: str*) → QModelIndex

Find the QModelIndex for the given model name.

is_condition_warning(*key: ConditionKey*) → bool

Return whether the condition key is currently flagged.

is_logic_warning(*key: LogicKey*) → bool

Return whether the logic key is currently flagged.

logic_key_for_index(*index: QModelIndex | QPersistentModelIndex*) → *LogicKey* | None

Return a logic key for the provided index.

logic_key_for_item(*item:* FilterTreeItem) → *LogicKey* | None

Return a logic key for the provided tree item.

set_condition_warning(*key:* ConditionKey, *enabled: bool*) → None

Enable or disable the warning icon for the provided condition key.

set_logic_warning(*key:* LogicKey, *enabled: bool*) → None

Enable or disable the warning icon for the provided logic key.

class mafw.steering_gui.models.filter_tree_model.**LogicKey**(*model_name: str | None*, *field_name: str | None*)

Bases: object

Stable key describing a logic entry in the filter tree.

mafw.steering_gui.models.filter_tree_model.**_RESOURCE_DIR = PosixPath('/tmp/mafw-docs-kpuvl5nc/v2.3.0_pdf/src/mafw/steering_gui/resources')**

Base path for GUI resources used by the filter tree.

mafw.steering_gui.models.filter_tree_model.**_WARNING_ICON_PATH = PosixPath('/tmp/mafw-docs-kpuvl5nc/v2.3.0_pdf/src/mafw/steering_gui/resources/mynaui--danger-triangle.svg')**

Icon path used to flag incomplete conditions.

mafw.steering_gui.models.pipeline

UI-only pipeline structures used to render editable processor hierarchies.

Author
Bulgheroni Antonio

Description
Lightweight in-memory objects that mirror the processor/group layout for the GUI.

Classes

PipelineItem(config, parent, children)	UI wrapper around a processor or group configuration.
ProcessorPipeline(items)	Container for the current processor pipeline tree.

class mafw.steering_gui.models.pipeline.**PipelineItem**(*config: ~mafw.steering.models.ProcessorConfig | ~mafw.steering.models.GroupConfig*, *parent: ~mafw.steering_gui.models.pipeline.PipelineItem | None = None*, *children: list[~mafw.steering_gui.models.pipeline.PipelineItem] = <factory>*)

Bases: object

UI wrapper around a processor or group configuration.

Parameters

- **config** – Domain configuration backing the item.
- **parent** – Parent pipeline item (None when top-level).
- **children** – Ordered list of child items.

is_group() → bool

Return True if this item wraps a group configuration.

name() → str

Return the replica-aware processor name or group name.

class mafw.steering_gui.models.pipeline.**ProcessorPipeline**(*items: list[~mafw.steering_gui.models.pipeline.PipelineItem] = <factory>*)

Bases: object

Container for the current processor pipeline tree.

Parameters
items – Top-level pipeline items in execution order.

mafw.steering_gui.models.processor_pipeline_model

Qt model for the processor pipeline tree.

Author
Bulgheroni Antonio

Description
Provide a hierarchical QAbstractItemModel view of pipeline items for GUI widgets.

Classes

PipelineRoles(*values)	Custom roles exported by the processor pipeline model.
ProcessorPipelineModel([pipeline, parent])	Expose a ProcessorPipeline as a tree-structured Qt model.

class mafw.steering_gui.models.processor_pipeline_model.**PipelineRoles**(**values*)

Bases: IntEnum

Custom roles exported by the processor pipeline model.

class mafw.steering_gui.models.processor_pipeline_model.**ProcessorPipelineModel**(*pipeline:* Processor-Pipeline | *None* = *None*, *parent: QObject* | *None* = *None*)

Bases: QAbstractItemModel

Expose a ProcessorPipeline as a tree-structured Qt model.

pipeline() → *ProcessorPipeline*

Return the currently attached pipeline.

root_items() → list[*PipelineItem*]

Return the top-level pipeline items.

set_pipeline(*pipeline:* ProcessorPipeline) → None

Replace the pipeline data and reset the model.

mafw.steering_gui.models.steering_tree_model

Composite Qt model that embeds the processor pipeline under the steering tree.

Author
Bulgheroni Antonio

Description
Presents the fixed steering sections plus the editable processors subtree.

Classes

SteeringTreeModel(controller[, pipeline, parent])	Tree model exposing the steering sections and embedded processor pipeline.
SteeringTreeRoles(*values)	Custom roles exported by the steering tree model.

class mafw.steering_gui.models.steering_tree_model.**SteeringTreeModel**(*controller: object*, *pipeline:* ProcessorPipeline | *None = None*, *parent: QObject | None = None*)

Bases: QAbstractItemModel

Tree model exposing the steering sections and embedded processor pipeline.

pipeline_item_from_index(*index: QModelIndex | QPersistentModelIndex*) → *PipelineItem* | None
Return the pipeline item stored at the given index, if any.

section_key_from_index(*index: QModelIndex | QPersistentModelIndex*) → str | None
Return the section key stored at the given index, if any.

set_pipeline(*pipeline:* ProcessorPipeline) → None
Replace the pipeline subtree and reset the model.

class mafw.steering_gui.models.steering_tree_model.**SteeringTreeRoles**(*values)

Bases: IntEnum

Custom roles exported by the steering tree model.

class mafw.steering_gui.models.steering_tree_model.**_SectionNode**(*label: str*, *key: str*, *children: list[~mafw.steering_gui.models.steering_tree_m = <factory>*)

Bases: object

Internal tree node for the fixed steering sections.

mafw.steering_gui.threading

Threading utilities made available to the steering GUI.

Author
Bulgheroni Antonio (antonio.bulgheroni@ec.europa.eu)

Description
Provide a shared Worker QObject for running operations off the GUI thread.

class mafw.steering_gui.threading.**Worker**(*fn: Callable[[...], T]*, **args: Any*, ***kwargs: Any*)

Bases: QRunnable, Generic[*T*]

Run a callable in a thread pool while exposing progress, error, and completion signals.

connect(*, *finished: Callable[[T], None] | None = None*, *error: Callable[[Exception], None] | None = None*, *progress: Callable[[int], None] | None = None*) → None

Attach callbacks to the worker signals.

run() → None

Execute the callable and propagate its result or any exception.

Modules

workers	Threading helpers for future steering GUI background tasks.

mafw.steering_gui.threading.workers

Threading helpers for future steering GUI background tasks.

Author
Bulgheroni Antonio (antonio.bulgheroni@ec.europa.eu)

Description
Define a reusable worker QObject that can execute callables without blocking the UI.

Module Attributes

T	Type variable for the return type of the worker's callable.

Classes

Worker(fn, *args, **kwargs)	Run a callable in a thread pool while exposing progress, error, and completion signals.
WorkerSignals	Bundle of signals emitted by a worker.

class mafw.steering_gui.threading.workers.**T**

Type variable for the return type of the worker's callable.

alias of TypeVar('T')

class mafw.steering_gui.threading.workers.**Worker**(*fn: Callable[[...], T]*, **args: Any*, ***kwargs: Any*)

Bases: QRunnable, Generic[*T*]

Run a callable in a thread pool while exposing progress, error, and completion signals.

connect(*, *finished: Callable[[T], None] | None = None*, *error: Callable[[Exception], None] | None = None*, *progress: Callable[[int], None] | None = None*) → None

Attach callbacks to the worker signals.

run() → None

Execute the callable and propagate its result or any exception.

class mafw.steering_gui.threading.workers.**WorkerSignals**

Bases: QObject, Generic[*T*]

Bundle of signals emitted by a worker.

mafw.steering_gui.utils

Helper utilities for the steering GUI.

mafw.steering_gui.utils.**block_signals**(**widgets: QObject*) → Iterator[None]

Temporarily disable Qt signals for a set of widgets.

mafw.steering_gui.utils.**coerce_name**(*text: str*) → str

Replace any whitespace in the provided text with underscores.

Modules

constants	Shared constants for the steering GUI.
exception_handling	GUI exception handling utilities for the steering application.
gui_helpers	Reusable GUI helpers shared across the steering GUI views.

mafw.steering_gui.utils.constants

Shared constants for the steering GUI.

Module Attributes

INT_RANGE	Default integer bounds for spin boxes.
FLOAT_RANGE	Default floating-point bounds for spin boxes.
FLOAT_DECIMALS	Default number of decimals for floating-point spin boxes.

mafw.steering_gui.utils.constants.**FLOAT_DECIMALS = 6**

Default number of decimals for floating-point spin boxes.

mafw.steering_gui.utils.constants.**FLOAT_RANGE = (-1000000000.0, 1000000000.0)**

Default floating-point bounds for spin boxes.

mafw.steering_gui.utils.constants.**INT_RANGE = (-2147483648, 2147483647)**

Default integer bounds for spin boxes.

mafw.steering_gui.utils.exception_handling

GUI exception handling utilities for the steering application.

Author

Bulgheroni Antonio (antonio.bulgheroni@ec.europa.eu)

Description

Provide a reusable PySide6-compatible exception handler that surfaces errors from the GUI and worker threads via dialog boxes.

Functions

get_thread_exception_handler()	Get or create the global thread exception handler.
install_exception_handler()	Install a global exception handler to show errors in a dialog box.
show_exception_dialog(exception_type, ...)	Display an exception in a message box dialog.

Classes

ThreadExceptionHandler([display_callback])	Handle exceptions raised in non-GUI threads and forward them to the GUI.

class mafw.steering_gui.utils.exception_handling.**ThreadExceptionHandler**(*display_callback: Callable[[type[BaseException] | None, BaseException | None, TracebackType | None], Any] | None = None*)

Bases: QObject

Handle exceptions raised in non-GUI threads and forward them to the GUI.

Initialize the handler and connect its signal to a display callback.

Parameters

display_callback (*Callable[[type[BaseException] | None, BaseException | None, TracebackType | None], Any]*) – Optional callable to show the exception details.

handle_exception(*exc_type: type[BaseException] | None*, *exc_value: BaseException | None*, *exc_traceback: TracebackType | None*) → None

Emit an exception signal so it can be shown on the GUI thread.

Parameters

- **exc_type** (*type[BaseException] | None*) – Type of the exception.
- **exc_value** (*BaseException | None*) – The exception instance.
- **exc_traceback** (*TracebackType | None*) – Traceback object.

mafw.steering_gui.utils.exception_handling.**get_thread_exception_handler**() → *ThreadExceptionHandler*

Get or create the global thread exception handler.

Returns

The global thread exception handler instance.

Return type

ThreadExceptionHandler

mafw.steering_gui.utils.exception_handling.**install_exception_handler**() → Callable[[type[BaseException], BaseException, TracebackType | None], Any]

Install a global exception handler to show errors in a dialog box.

Returns

The original sys.excepthook implementation.

Return type

Callable[[type[BaseException], BaseException, TracebackType | None], Any]

mafw.steering_gui.utils.exception_handling.**show_exception_dialog**(*exception_type: type[BaseException] | None*, *exception_value: BaseException | None*, *exception_traceback: TracebackType | None*) → None

Display an exception in a message box dialog.

Parameters

- **exception_type** (*type[BaseException] | None*) – Type of the exception.
- **exception_value** (*BaseException | None*) – The exception instance.
- **exception_traceback** (*TracebackType | None*) – Traceback object.

mafw.steering_gui.utils.exception_handling.**_DisplayCallback**

Type alias for exception display callbacks.

alias of `Callable[[type[BaseException] | None, BaseException | None, TracebackType | None], Any]`

mafw.steering_gui.utils.exception_handling.**_thread_exception_handler: *ThreadExceptionHandler* | None = None**

Global instance of the thread exception handler.

mafw.steering_gui.utils.gui_helpers

Reusable GUI helpers shared across the steering GUI views.

Functions

block_signals(*widgets)	Temporarily disable Qt signals for a set of widgets.
coerce_name(text)	Replace any whitespace in the provided text with underscores.

mafw.steering_gui.utils.gui_helpers.**block_signals**(**widgets: QObject*) → Iterator[None]

Temporarily disable Qt signals for a set of widgets.

mafw.steering_gui.utils.gui_helpers.**coerce_name**(*text: str*) → str

Replace any whitespace in the provided text with underscores.

mafw.steering_gui.views

Placeholder package for steering GUI views while the GUI is being scaffolded.

Author
Bulgheroni Antonio (antonio.bulgheroni@ec.europa.eu)

Description
UI views live here once they are implemented, keeping the controller agnostic of toolkit code.

class mafw.steering_gui.views.**DatabaseEditor**(*parent: QWidget | None = None*)

Bases: `QWidget`

Widget that exposes database backend selection and layered configuration.

_detect_backend_from_url(*url: str | None*) → str

Determine the backend type from the given URL.

This method analyses the provided URL to identify the corresponding backend type by matching URL prefixes or extracting the protocol. If no match is found, it defaults to the current backend or the default backend.

Parameters
url – The URL string to be analysed for backend detection.

Returns
A string representing the detected backend type.

_normalize_config(*config: dict[str, Any] | None*) → dict[str, Any]

Normalize the configuration dictionary to ensure consistent backend, URL, authentication, and parameters.

This method processes the input configuration dictionary to extract and normalize values for backend type, URL, authentication details, and backend-specific parameters, ensuring defaults are applied where necessary.

Parameters
config – A dictionary containing configuration details such as backend, URL, authentication, and parameters. If None, defaults are applied.

Returns
A dictionary with normalized configuration values including backend, URL, enabled status, authentication, parameters, and pragmas.

get_data() → dict[str, Any]

Return a normalized view of the current database configuration.

is_enabled() → bool

Return whether the database configuration group is active.

set_data(*config: dict[str, Any] | None*) → None

Populate the UI from a configuration dictionary without emitting signals.

class mafw.steering_gui.views.**GlobalsEditor**(*parent: QWidget | None = None*)

Bases: QWidget

Editor that exposes analysis metadata and global flags without touching the builder.

current_values() → tuple[str, str, bool, bool]

Return the current UI values without normalizing them.

set_data(*analysis_name: str | None*, *description: str | None*, *new_only: bool | None*, *create_standard_tables: bool | None*) → None

Populate the widgets and avoid re-triggering their change signals.

class mafw.steering_gui.views.**GroupEditor**(*parent: QWidget | None = None*)

Bases: QWidget

Widget used to edit group metadata and the processor list.

refresh() → None

Refresh the table root index after model updates.

set_group_item(*item:* PipelineItem) → None

Display the provided group item and its processors.

set_model(*model:* SteeringTreeModel) → None

Attach the steering tree model to the group editor.

class mafw.steering_gui.views.**MetadataEditor**(*parent: QWidget | None = None*)

Bases: QWidget

Provide a quick summary of the steering file when no specific section is active.

set_validation_level(*level:* ValidationLevel | *None*) → None

Update the combo to reflect the controller's validation preference.

update_summary(*analysis_name: str | None*, *processor_count: int*, *available_processors: int | None = None*, *available_db_modules: int | None = None*, *available_uis: int | None = None*) → None

Refresh the summary labels to match the loaded model.

class mafw.steering_gui.views.**PipelineEditor**(*parent: QWidget | None = None*)

Bases: QWidget

Widget exposing the processor pipeline as a flat table plus toolbox actions.

refresh() → None

Refresh the processors root index after model resets.

set_model(*model:* SteeringTreeModel) → None

Attach the steering tree model and focus on the processors section.

class mafw.steering_gui.views.**ProcessorEditor**(*parent: QWidget | None = None*)

Bases: QWidget

Placeholder widget representing per-processor configuration editors.

class mafw.steering_gui.views.**ProcessorParameterEditor**(*parent: QWidget | None = None*)

Bases: QWidget

Widget used to edit parameters and metadata of a processor entry.

current_name() → str | None

Return the currently displayed processor full name.

set_processor_data(*config:* ProcessorConfig) → None

Populate the editor with processor configuration data.

class mafw.steering_gui.views.**SteeringTreeView**(*parent: QWidget | None = None*)

Bases: QWidget

Widget exposing the steering-file sections as a QTreeView.

_on_selection_changed(**_: object*) → None

Emit the section key and pipeline selection for the new tree selection.

refresh() → None

Refresh expansion and selection after model updates.

select_pipeline_item(*name: str*) → None

Select a pipeline item by its processor/group name.

set_model(*model:* SteeringTreeModel) → None

Attach a steering tree model to the view.

class mafw.steering_gui.views.**UIEditor**(*parent: QWidget | None = None*)

Bases: QWidget

Widget that exposes the configured user interface and allows switching it.

set_available_interfaces(*names: Iterable[str] | None*) → None

Populate the combo box once plugins have produced their UI list.

set_current_interface(*interface: str | None*) → None

Request the combobox to display a specific interface without emitting signals.

Modules

database_editor	Expose the database editor UI for steering file configuration.
filter_editors	Editors for different filter types in the steering GUI.
filter_manager	Manager widget for filter configuration.
filter_tree	Tree view for displaying and interacting with processor filters.
globals_editor	Expose the globals editor fields required by the steering GUI.
group_editor	Group editor widget for managing processor groups.
meta_editor	Editor showing metadata when the root node is selected.
pipeline_editor	Pipeline editor widget exposing processors and groups in execution order.
processor_editor	Editor scaffolding for processor-specific configurations.
processor_parameter_editor	Processor parameter editor for steering GUI pipelines.
steering_tree	Present a hierarchical representation of the steering file sections.
ui_editor	Editor for selecting the user interface defined in a steering file.

mafw.steering_gui.views.database_editor

Expose the database editor UI for steering file configuration.

Author
Bulgheroni Antonio (antonio.bulgheroni@ec.europa.eu)

Description
Provide a widget to edit database backend, transport, authentication, and backend parameters without touching the builder.

Module Attributes

DEFAULT_BACKEND	Default backend selection for the database editor.
DEFAULT_URLS	Mapping of backend identifiers to URL prefixes.
DEFAULT_PRAGMAS	Default SQLite pragma values used for parameter seeding.
DEFAULT_SERVER_PORTS	Default service ports used by server backends.
DEFAULT_URL_PLACEHOLDERS	Placeholder URL examples for each backend.

Classes

AuthenticationWidget([parent])	Widget exposing authentication configuration for server databases.
DatabaseEditor([parent])	Widget that exposes database backend selection and layered configuration.
ParametersWidget([parent])	Widget that manages backend-specific parameters.
TransportWidget([parent])	Widget exposing the database transport layer (URL).

class mafw.steering_gui.views.database_editor.**AuthenticationWidget**(*parent: QWidget | None = None*)

Bases: QWidget

Widget exposing authentication configuration for server databases.

get_data() → dict[str, Any]

Return the authentication mapping for the current method.

has_required_fields() → bool

Return whether required fields for the selected method are populated.

set_backend(*backend: str*) → None

Enable or disable auth inputs based on backend.

set_data(*auth: Mapping[str, Any] | None*) → None

Populate widget values from the auth mapping.

property file_browse_button: QToolButton

Expose browse button for tests.

property file_edit: QLineEdit

Expose password file edit for tests.

property method_combo: QComboBox

Expose method combo for tests.

property password_edit: QLineEdit

Expose password edit for tests.

property username_edit: QLineEdit

Expose username edit for tests.

class mafw.steering_gui.views.database_editor.**DatabaseEditor**(*parent: QWidget | None = None*)

Bases: QWidget

Widget that exposes database backend selection and layered configuration.

_detect_backend_from_url(*url: str | None*) → str

Determine the backend type from the given URL.

This method analyses the provided URL to identify the corresponding backend type by matching URL prefixes or extracting the protocol. If no match is found, it defaults to the current backend or the default backend.

Parameters

url – The URL string to be analysed for backend detection.

Returns

A string representing the detected backend type.

_normalize_config(*config: dict[str, Any] | None*) → dict[str, Any]

Normalize the configuration dictionary to ensure consistent backend, URL, authentication, and parameters.

This method processes the input configuration dictionary to extract and normalize values for backend type, URL, authentication details, and backend-specific parameters, ensuring defaults are applied where necessary.

Parameters

config – A dictionary containing configuration details such as backend, URL, authentication, and parameters. If None, defaults are applied.

Returns

A dictionary with normalized configuration values including backend, URL, enabled status, authentication, parameters, and pragmas.

get_data() → dict[str, Any]

Return a normalized view of the current database configuration.

is_enabled() → bool

Return whether the database configuration group is active.

set_data(*config: dict[str, Any] | None*) → None

Populate the UI from a configuration dictionary without emitting signals.

class mafw.steering_gui.views.database_editor.**ParametersWidget**(*parent: QWidget | None = None*)

Bases: QWidget

Widget that manages backend-specific parameters.

current_table_values() → dict[str, Any]

Return the current backend table mapping.

get_data() → dict[str, dict[str, Any]]

Return the full backend parameter mapping.

set_backend(*backend: str*) → None

Switch the visible parameter table to the given backend.

set_data(*parameters: Mapping[str, Mapping[str, Any]] | None*, *backend: str*) → None

Populate the widget with the provided backend parameter mapping.

property table: QTableView

Expose the parameter table for tests.

class mafw.steering_gui.views.database_editor.**TransportWidget**(*parent: QWidget | None = None*)

Bases: QWidget

Widget exposing the database transport layer (URL).

get_url() → str

Return the current URL string.

is_valid() → bool

Return whether the URL can be parsed.

set_backend(*backend: str*) → None

Switch the transport view according to the backend.

set_url(*url: str*) → None

Update the URL field without emitting signals.

property sqlite_memory_button: QToolButton

Expose the SQLite memory button for tests.

property sqlite_open_button: QToolButton

Expose the SQLite open button for tests.

property url_edit: QLineEdit

Expose the active URL edit widget for external tests.

mafw.steering_gui.views.database_editor.**DEFAULT_BACKEND = 'sqlite'**

Default backend selection for the database editor.

mafw.steering_gui.views.database_editor.**DEFAULT_PRAGMAS: dict[str, Any] = {'cache_size': -64000, 'foreign_keys': 1, 'journal_mode': 'wal', 'synchronous': 0}**

Default SQLite pragma values used for parameter seeding.

mafw.steering_gui.views.database_editor.**DEFAULT_SERVER_PORTS: dict[str, int] = {'mysql': 3306, 'postgresql': 5432}**

Default service ports used by server backends.

mafw.steering_gui.views.database_editor.**DEFAULT_URLS: dict[str, str] = {'mysql': 'mysql://', 'postgresql': 'postgresql://', 'sqlite': 'sqlite:///'}**

Mapping of backend identifiers to URL prefixes.

mafw.steering_gui.views.database_editor.**DEFAULT_URL_PLACEHOLDERS: dict[str, str] = {'mysql': 'mysql://localhost:3306/mydb', 'postgresql': 'postgresql://localhost:5432/mydb', 'sqlite': 'sqlite:////path/to/db'}**

Placeholder URL examples for each backend.

mafw.steering_gui.views.filter_editors

Editors for different filter types in the steering GUI.

Author

Bulgheroni Antonio (antonio.bulgheroni@ec.europa.eu)

Description

Collection of widgets for editing model, field, condition, conditional, and logic filters.

Classes

ConditionDefinition([parent])	Reusable widget containing the core condition definition controls: Field/Name selection, Operator selection, and Value input.
ConditionEditor([parent])	Editor for Condition.
ConditionalFilterEditor([parent])	Editor for ConditionalFilterConfig.
ConditionsTableModel(config[, parent])	Table model for displaying conditions in a FieldFilterConfig.
FieldFilterEditor([parent])	Editor for FieldFilterConfig.
FieldNameWidget([parent])	Name editor for model-level conditions based on model fields.
FreeNameWidget(model_name, field_name[, parent])	Name editor for field-level conditions with free text.
ListEditor([parent])	Editor for list-based conditions.
ListValueRow(field_type[, parent])	Row widget for list-based values.
LogicEditor([parent])	Editor for Logic expressions.
LogicSyntaxHighlighter([parent])	Syntax highlighter for logic expressions.
LogicTextEdit([parent])	Plain text editor with QCompleter support for logic conditions.
ModelFieldsTableModel(model_info, ...[, parent])	Table model for displaying model fields and their filters.
ModelFilterEditor([parent])	Editor for ModelFilterConfig.
ProcessorFilterEditor([parent])	Editor for processor-level filter configuration.
ProcessorModelsTableModel(processor_config, ...)	Table model for displaying available processor models.
RangeEditor([parent])	Editor for BETWEEN conditions.
SingleValueEditor([parent])	Editor for a single value condition.
ValueEditorBase	Base class for condition value editors.

class mafw.steering_gui.views.filter_editors.**ConditionDefinition**(*parent: QWidget | None = None*)

Bases: QWidget

Reusable widget containing the core condition definition controls: Field/Name selection, Operator selection, and Value input.

is_complete() → bool

Return whether the current condition has a valid value selection.

set_condition(*condition:* Condition, *parent:* ModelFilterConfig | FieldFilterConfig | ConditionalFilterConfig, *condition_name: str*) → None

Bind the editor to the provided condition.

class mafw.steering_gui.views.filter_editors.**ConditionEditor**(*parent: QWidget | None = None*)

Bases: QWidget

Editor for Condition.

is_complete() → bool

Return whether the current condition has a valid value selection.

set_condition(*condition:* Condition, *parent:* ModelFilterConfig | FieldFilterConfig | ConditionalFilterConfig, *condition_name: str*) → None

Bind the editor to the provided condition.

class mafw.steering_gui.views.filter_editors.**ConditionalFilterEditor**(*parent: QWidget | None = None*)

Bases: QWidget

Editor for ConditionalFilterConfig.

class mafw.steering_gui.views.filter_editors.**ConditionsTableModel**(*config:* FieldFilterConfig, *parent: QWidget | None = None*)

Bases: QAbstractTableModel

Table model for displaying conditions in a FieldFilterConfig.

class mafw.steering_gui.views.filter_editors.**FieldFilterEditor**(*parent: QWidget | None = None*)

Bases: QWidget

Editor for FieldFilterConfig.

set_config(*data:* FieldFilterConfig | *str*, *processor_config:* ProcessorConfig, *index: QModelIndex*) → None

Update the editor with configuration data.

class mafw.steering_gui.views.filter_editors.**FieldNameWidget**(*parent: QWidget | None = None*)

Bases: QWidget

Name editor for model-level conditions based on model fields.

get_name() → str

Return the selected field name.

get_type() → type

Return the python type for the selected field.

set_existing_names(*names: Iterable[str]*, *current: str | None*) → None

Filter out names already used by sibling conditions.

set_model_info(*model_info:* ModelInfo | *None*) → None

Populate the combo box with fields from the provided model info.

set_name(*name: str*) → None

Set the selected field name.

class mafw.steering_gui.views.filter_editors.**FreeNameWidget**(*model_name: str*, *field_name: str*, *parent: QWidget | None = None*)

Bases: QWidget

Name editor for field-level conditions with free text.

get_name() → str

Return the entered condition name.

get_type() → type

Return the python type for the configured field.

matches(*model_name: str*, *field_name: str*) → bool

Return whether the widget matches the provided context.

set_name(*name: str*) → None

Set the displayed condition name.

update_context(*model_name: str*, *field_name: str*) → None

Update the model/field context used for typing.

class mafw.steering_gui.views.filter_editors.**ListEditor**(*parent: QWidget | None = None*)

Bases: *ValueEditorBase*

Editor for list-based conditions.

get_value() → list[Any]

Return the current value.

set_field_type(*field_type: type*) → None

Update the editor to reflect the provided field type.

set_value(*value: Any*) → None

Set the current value.

class mafw.steering_gui.views.filter_editors.**ListValueRow**(*field_type: type*, *parent: QWidget | None = None*)

Bases: QWidget

Row widget for list-based values.

class mafw.steering_gui.views.filter_editors.**LogicEditor**(*parent: QWidget | None = None*)

Bases: QWidget

Editor for Logic expressions.

is_valid() → bool

Return whether the current logic expression is valid.

set_config(*data:* ProcessorConfig | ModelFilterConfig | FieldFilterConfig, *processor_config:* ProcessorConfig, *index: QModelIndex*) → None

Bind the editor to the provided configuration data.

class mafw.steering_gui.views.filter_editors.**LogicSyntaxHighlighter**(*parent: Any = None*)

Bases: QSyntaxHighlighter

Syntax highlighter for logic expressions.

class mafw.steering_gui.views.filter_editors.**LogicTextEdit**(*parent: QWidget | None = None*)

Bases: QPlainTextEdit

Plain text editor with QCompleter support for logic conditions.

class mafw.steering_gui.views.filter_editors.**ModelFieldsTableModel**(*model_info: ModelInfo*, *model_config: ModelFilterConfig | None*, *processor_config: ProcessorConfig*, *parent: QWidget | None = None*)

Bases: QAbstractTableModel

Table model for displaying model fields and their filters.

class mafw.steering_gui.views.filter_editors.**ModelFilterEditor**(*parent: QWidget | None = None*)

Bases: QWidget

Editor for ModelFilterConfig.

set_config(*data: ModelFilterConfig | str*, *processor_config: ProcessorConfig*, *index: QModelIndex*) → None

Update the editor with configuration data.

class mafw.steering_gui.views.filter_editors.**ProcessorFilterEditor**(*parent: QWidget | None = None*)

Bases: QWidget

Editor for processor-level filter configuration.

set_config(*config: ProcessorConfig*) → None

Update the editor with processor configuration data.

class mafw.steering_gui.views.filter_editors.**ProcessorModelsTableModel**(*processor_config: ProcessorConfig*, *advertised_models: list[str]*, *all_models: list[str]*, *parent: QWidget | None = None*)

Bases: QAbstractTableModel

Table model for displaying available processor models.

get_model_name(*row: int*) → str

Return the model name for the given row.

class mafw.steering_gui.views.filter_editors.**RangeEditor**(*parent: QWidget | None = None*)

Bases: *ValueEditorBase*

Editor for BETWEEN conditions.

get_value() → list[Any]

Return the current value.

set_field_type(*field_type: type*) → None

Update the editor to reflect the provided field type.

set_value(*value: Any*) → None

Set the current value.

class mafw.steering_gui.views.filter_editors.**SingleValueEditor**(*parent: QWidget | None = None*)

Bases: *ValueEditorBase*

Editor for a single value condition.

get_value() → Any
Return the current value.

set_field_type(*field_type: type*) → None
Update the editor to reflect the provided field type.

set_value(*value: Any*) → None
Set the current value.

class mafw.steering_gui.views.filter_editors.**ValueEditorBase**
Bases: QWidget

Base class for condition value editors.

get_value() → Any
Return the current value.

set_field_type(*field_type: type*) → None
Update the editor to reflect the provided field type.

set_value(*value: Any*) → None
Set the current value.

mafw.steering_gui.views.filter_editors.**_create_value_widget**(*field_type: type*, *parent: QWidget | None = None*) → QWidget
Create an editor widget suitable for the provided field type.

mafw.steering_gui.views.filter_editors.**_get_widget_value**(*widget: QWidget*) → Any
Return the current value from a value widget.

mafw.steering_gui.views.filter_editors.**_operators_for_type**(*field_type: type*) → list[*LogicalOp*]
Return the allowed operators for the given field type.

mafw.steering_gui.views.filter_editors.**_set_widget_value**(*widget: QWidget*, *value: Any*) → None
Set the current value on a value widget.

mafw.steering_gui.views.filter_editors.**_NO_VALUE_OPS = {LogicalOp.IS_NOT_NULL, LogicalOp.IS_NULL}**
Operators that do not require a value input.

mafw.steering_gui.views.filter_editors.**_OPERATOR_TOOLTIPS: dict[*LogicalOp*, str] = {LogicalOp.BETWEEN: 'Between the provided bounds', LogicalOp.BIT_AND: 'Bitwise AND against the provided value', LogicalOp.BIT_OR: 'Bitwise OR against the provided value', LogicalOp.EQ: 'Equal to', LogicalOp.GE: 'Greater than or equal to', LogicalOp.GLOB: 'Glob-style pattern matching', LogicalOp.GT: 'Greater than', LogicalOp.IN: 'Contained in the provided list', LogicalOp.IS_NOT_NULL: 'Value is not NULL', LogicalOp.IS_NULL: 'Value is NULL', LogicalOp.LE: 'Less than or equal to', LogicalOp.LIKE: 'SQL LIKE pattern matching', LogicalOp.LT: 'Less than', LogicalOp.NE: 'Not equal to', LogicalOp.NOT_IN: 'Not contained in the provided list', LogicalOp.REGEXP: 'Regular expression matching'}**
Tooltip text for each logical operator.

mafw.steering_gui.views.filter_manager

Manager widget for filter configuration.

Author
Bulgheroni Antonio (antonio.bulgheroni@ec.europa.eu)

Description
Combines a FilterTreeView and a stack of editors to manage processor filters.

Classes

FilterManager(config[, parent])	Widget for managing processor filters.

class mafw.steering_gui.views.filter_manager.**FilterManager**(*config: ProcessorConfig*, *parent: QWidget | None = None*)

Bases: QWidget

Widget for managing processor filters.

_on_conditional_filter_changed() → None

Handle changes in the ConditionalFilterEditor.

refresh() → None

Refresh the tree model.

mafw.steering_gui.views.filter_tree

Tree view for displaying and interacting with processor filters.

Author
Bulgheroni Antonio (antonio.bulgheroni@ec.europa.eu)

Description
QTreeView implementation that provides context menus for adding and removing filter components.

Classes

FilterTreeView([parent])	Tree view for managing filter hierarchy.

class mafw.steering_gui.views.filter_tree.**FilterTreeView**(*parent: QWidget | None = None*)

Bases: QTreeView

Tree view for managing filter hierarchy.

_first_unused_model_field(*model_name: str*, *existing: Iterable[str]*) → str | None

Return the first model field without a configured condition.

_has_logic(*config: ProcessorConfig | ModelFilterConfig | FieldFilterConfig*) → bool

Return whether the configuration has logic defined or buffered.

add_model_filter_by_name(*model_name: str*) → None

Add a ModelFilterConfig for the given model name.

add_root_logic() → None

Add or edit processor-level logic.

remove_item(*index: QModelIndex*) → bool

Remove the item at the provided index.

Returns:
True if the item was removed, False otherwise.

set_model(*model: FilterTreeModel*) → None

Set the model and expand all nodes.

mafw.steering_gui.views.globals_editor

Expose the globals editor fields required by the steering GUI.

Author
Bulgheroni Antonio (antonio.bulgheroni@ec.europa.eu)

Description
Provide input fields for the analysis metadata and flags managed by the controller.

Classes

GlobalsEditor([parent])	Editor that exposes analysis metadata and global flags without touching the builder.

class mafw.steering_gui.views.globals_editor.**GlobalsEditor**(*parent: QWidget | None = None*)

Bases: QWidget

Editor that exposes analysis metadata and global flags without touching the builder.

current_values() → tuple[str, str, bool, bool]

Return the current UI values without normalizing them.

set_data(*analysis_name: str | None*, *description: str | None*, *new_only: bool | None*, *create_standard_tables: bool | None*) → None

Populate the widgets and avoid re-triggering their change signals.

mafw.steering_gui.views.group_editor

Group editor widget for managing processor groups.

Author
Bulgheroni Antonio (antonio.bulgheroni@ec.europa.eu)

Description
Provide a table-based editor to rename groups, set descriptions, and reorder processors or nested groups without direct controller access.

Classes

GroupEditor([parent])	Widget used to edit group metadata and the processor list.

class mafw.steering_gui.views.group_editor.**GroupEditor**(*parent: QWidget | None = None*)

Bases: QWidget

Widget used to edit group metadata and the processor list.

refresh() → None

Refresh the table root index after model updates.

set_group_item(*item:* PipelineItem) → None

Display the provided group item and its processors.

set_model(*model:* SteeringTreeModel) → None

Attach the steering tree model to the group editor.

class mafw.steering_gui.views.group_editor.**_GroupTableView**

Bases: QTableView

Internal table view that exposes delete key handling.

mafw.steering_gui.views.group_editor.**_DEFAULT_DESCRIPTION = ''**

Fallback description used when none is configured.

mafw.steering_gui.views.group_editor.**_DEFAULT_GROUP_NAME = 'Group'**

Base name used when presenting an unnamed group.

mafw.steering_gui.views.meta_editor

Editor showing metadata when the root node is selected.

Author
Bulgheroni Antonio (antonio.bulgheroni@ec.europa.eu)

Description
Give users contextual guidance when the entire steering file is selected.

Classes

MetadataEditor([parent])	Provide a quick summary of the steering file when no specific section is active.

class mafw.steering_gui.views.meta_editor.**MetadataEditor**(*parent: QWidget | None = None*)

Bases: QWidget

Provide a quick summary of the steering file when no specific section is active.

set_validation_level(*level:* ValidationLevel | *None*) → None

Update the combo to reflect the controller's validation preference.

update_summary(*analysis_name: str | None*, *processor_count: int*, *available_processors: int | None = None*, *available_db_modules: int | None = None*, *available_uis: int | None = None*) → None

Refresh the summary labels to match the loaded model.

mafw.steering_gui.views.pipeline_editor

Pipeline editor widget exposing processors and groups in execution order.

Author
Bulgheroni Antonio (antonio.bulgheroni@ec.europa.eu)

Description
Provide a table-based editor to reorder, edit, and add pipeline entries without direct controller access.

Classes

PipelineEditor([parent])	Widget exposing the processor pipeline as a flat table plus toolbox actions.

class mafw.steering_gui.views.pipeline_editor.**PipelineEditor**(*parent: QWidget | None = None*)

Bases: QWidget

Widget exposing the processor pipeline as a flat table plus toolbox actions.

refresh() → None

Refresh the processors root index after model resets.

set_model(*model:* SteeringTreeModel) → None

Attach the steering tree model and focus on the processors section.

class mafw.steering_gui.views.pipeline_editor.**_PipelineTableView**

Bases: QTableView

Internal table view that exposes delete key handling.

mafw.steering_gui.views.processor_editor

Editor scaffolding for processor-specific configurations.

Author
Bulgheroni Antonio (antonio.bulgheroni@ec.europa.eu)

Description
Offer a placeholder layout that will later host processor settings.

Classes

ProcessorEditor([parent])	Placeholder widget representing per-processor configuration editors.

class mafw.steering_gui.views.processor_editor.**ProcessorEditor**(*parent: QWidget | None = None*)

Bases: QWidget

Placeholder widget representing per-processor configuration editors.

mafw.steering_gui.views.processor_parameter_editor

Processor parameter editor for steering GUI pipelines.

Author
Bulgheroni Antonio (antonio.bulgheroni@ec.europa.eu)

Description
Display and edit processor parameters, replica metadata, and overrides in the GUI.

Classes

ParameterDelegate	Delegate that picks editors based on parameter type.
ParameterTableModel([parent])	Qt table model exposing processor parameters for editing.
PathEditor([parent])	Inline editor combining a line edit with a browse button.
ProcessorParameterEditor([parent])	Widget used to edit parameters and metadata of a processor entry.

class mafw.steering_gui.views.processor_parameter_editor.**ParameterDelegate**

Bases: QStyledItemDelegate

Delegate that picks editors based on parameter type.

class mafw.steering_gui.views.processor_parameter_editor.**ParameterTableModel**(*parent: QWidget | None = None*)

Bases: QAbstractTableModel

Qt table model exposing processor parameters for editing.

set_parameters(*parameters: Iterable[ParameterConfig]*) → None

Replace the rows with the provided parameters.

class mafw.steering_gui.views.processor_parameter_editor.**PathEditor**(*parent: QWidget | None = None*)

Bases: QWidget

Inline editor combining a line edit with a browse button.

set_value(*value: Any*) → None

Update the displayed path string.

value() → str

Return the currently selected path string.

class mafw.steering_gui.views.processor_parameter_editor.**ProcessorParameterEditor**(*parent: QWidget | None = None*)

Bases: QWidget

Widget used to edit parameters and metadata of a processor entry.

current_name() → str | None

Return the currently displayed processor full name.

set_processor_data(*config: ProcessorConfig*) → None

Populate the editor with processor configuration data.

class mafw.steering_gui.views.processor_parameter_editor.**_ParameterRoles**

Bases: object

Custom roles used to expose parameter metadata to delegates.

class mafw.steering_gui.views.processor_parameter_editor.**_ParameterRow**(*config: ParameterConfig, order: int*)

Bases: object

Container describing a single parameter table row.

mafw.steering_gui.views.processor_parameter_editor.**_ACTIVE_COLUMN_LABEL = 'Active'**

Header label for the parameter active toggle column.

mafw.steering_gui.views.processor_parameter_editor.**_COLUMNS = ('Active', 'Name', 'Value', 'Type', 'Source', 'Status')**

Column labels for the processor parameter table.

mafw.steering_gui.views.steering_tree

Present a hierarchical representation of the steering file sections.

Author

Bulgheroni Antonio (antonio.bulgheroni@ec.europa.eu)

Description
Show globals, database, UI, and processors in a tree so the user can select editors.

Classes

SteeringTreeView([parent])	Widget exposing the steering-file sections as a `QTreeView`.

class mafw.steering_gui.views.steering_tree.**SteeringTreeView**(*parent: QWidget | None = None*)

Bases: `QWidget`

Widget exposing the steering-file sections as a `QTreeView`.

_on_selection_changed(**_: object*) → None
Emit the section key and pipeline selection for the new tree selection.

refresh() → None
Refresh expansion and selection after model updates.

select_pipeline_item(*name: str*) → None
Select a pipeline item by its processor/group name.

set_model(*model:* SteeringTreeModel) → None
Attach a steering tree model to the view.

mafw.steering_gui.views.ui_editor

Editor for selecting the user interface defined in a steering file.

Author
Bulgheroni Antonio (antonio.bulgheroni@ec.europa.eu)

Description
Allow the user to pick which UI plugin should drive execution.

Classes

UIEditor([parent])	Widget that exposes the configured user interface and allows switching it.

class mafw.steering_gui.views.ui_editor.**UIEditor**(*parent: QWidget | None = None*)

Bases: `QWidget`

Widget that exposes the configured user interface and allows switching it.

set_available_interfaces(*names: Iterable[str] | None*) → None
Populate the combo box once plugins have produced their UI list.

set_current_interface(*interface: str | None*) → None
Request the combobox to display a specific interface without emitting signals.

14.1.19 mafw.timer

Module implements a simple timer to measure the execution duration.

Basic usage:

```
from mafw import timer

with Timer() as timer:
    do_long_lasting_operation()
```

When exiting from the context manager, a message with the duration of the process is printed.

Functions

pretty_format_duration(duration_s[, n_digits])	Return a formatted version of the duration with increased human readability.
rreplace(inp_string, old_string, new_string, ...)	Utility function to replace a substring in a given string a certain number of times starting from the right-most one.

Classes

Timer([suppress_message])	The timer class.

class mafw.timer.**Timer**(*suppress_message: bool = False*)

Bases: `object`

The timer class.

Constructor parameter:

Parameters
suppress_message (*bool*) – A boolean flag to mute the timer

format_duration() → str

Nicely format the timer duration.

Returns
A string with the timer duration in a human-readable formatted string

Return type
str

property duration: float

The elapsed time of the timer.

Returns
Elapsed time in seconds.

mafw.timer.**pretty_format_duration**(*duration_s: float*, *n_digits: int = 1*) → str

Return a formatted version of the duration with increased human readability.

Parameters

- **duration_s** (*float*) – The duration to be printed in seconds. If negative, a ValueError exception is raised.
- **n_digits** (*int, Optional*) – The number of decimal digits to show. Defaults to 1. If negative, a ValueError exception is raised.

Returns
The formatted string.

Return type
str

Raises
ValueError – if a negative duration or a negative number of digits is provided

mafw.timer.**rreplace**(*inp_string: str*, *old_string: str*, *new_string: str*, *counts: int*) → str

Utility function to replace a substring in a given string a certain number of times starting from the right-most one.

This function is mimicking the behavior of the string.replace method, but instead of replacing from the left, it is replacing from the right.

Parameters

- **inp_string** (*str*) – The input string
- **old_string** (*str*) – The old substring to be matched. If empty, a ValueError is raised.
- **new_string** (*str*) – The new substring to be replaced
- **counts** (*int*) – The number of times the old substring has to be replaced.

Returns
The modified string

Return type
str

Raises
ValueError – if old_string is empty.

14.1.20 mafw.tools

The package provides a set of tools for different range of applications.

Modules

click_extensions	Reusable Click group classes and shell completion tools for MAFw command-line interfaces.
db_tools	Database Tools
file_tools	The module provides utilities for handling file, filename, hashing and so on.
generics	
loop_tuning	Mixin utilities for tuning loop execution parameters via steering files.
pandas_tools	A collection of useful convenience functions for common pandas operations
parallel	Utilities for parallel execution in MAFw.
regexp	Module implements some basic functions involving regular expressions.
shell_tools	Shell execution utilities.
toml_tools	Tools for reading, writing, and validating MAFw TOML steering files.

mafw.tools.click_extensions

Reusable Click group classes and shell completion tools for MAFw command-line interfaces.

This module centralizes the command abbreviation behavior used by the `mafw` executable and the development tools so nested command groups can inherit the same resolution policy without repeating the `cls=...` configuration. It also provides common helper functions for shell completion installation and management across all MAFw CLI tools.

Added in version 2.2.

Authors

Bulgheroni Antonio <antonio.bulgheroni@ec.europa.eu>

Functions

check_ci_completion_guard()	Check if the current environment is a CI environment.
completion_script_path(tool_name, shell)	Build the completion script path inside the active virtual environment.
completion_shell_from_env(shell_path)	Resolve a completion shell from `$SHELL`.
completion_source_script(tool_name, shell)	Generate the Click completion source script for the requested shell.
install_completion(tool_name, shell, force, ...)	Install Click completion for the requested shell.
is_script_already_installed(tool_name, shell)	Check if the completion script for the tool is already installed.
resolve_completion_shell(shell)	Normalize the requested completion shell.
uninstall_completion_files(tool_name[, shell])	Remove installed completion files and activation hooks.

Classes

AbbreviateGroup([name, commands, ...])	Click group that resolves unique command prefixes and catches DevtoolsError.
DeprecatedOption(*args[, deprecated_message])	A Click Option subclass that emits a DeprecationWarning when explicitly used.

class mafw.tools.click_extensions.**AbbreviateGroup**(*name: str | None = None*, *commands: MutableMapping[str, Command] | Sequence[Command] | None = None*, *invoke_without_command: bool = False*, *no_args_is_help: bool | None = None*, *subcommand_metavar: str | None = None*, *chain: bool = False*, *result_callback: Callable[[...], Any] | None = None*, ***kwargs: Any*)

Bases: `Group`

Click group that resolves unique command prefixes and catches DevtoolsError.

group_class

alias of *AbbreviateGroup*

get_command(*ctx: Context*, *cmd_name: str*) → Command | None

Return a command by exact name or a unique abbreviation.

invoke(*ctx: Context*) → Any

Invoke the group, catching DevtoolsError and converting to ClickException.

resolve_command(*ctx: Context*, *args: list[str]*) → tuple[str | None, Command | None, list[str]]

Return the canonical command name for abbreviated commands.

class mafw.tools.click_extensions.**DeprecatedOption**(**args: Any*, *deprecated_message: str = ''*, ***kwargs: Any*)

Bases: `Option`

A Click Option subclass that emits a DeprecationWarning when explicitly used.

Use this class as `cls=DeprecatedOption` in a `@click.option` decorator to mark an option as deprecated. The warning is only emitted when the user explicitly provides the option on the command line; default-value resolution does **not** trigger the warning. The resolved value is passed through to the command callback unchanged.

Parameters
deprecated_message (*str*) – Warning text emitted when the option is explicitly provided on the command line.

Added in version 2.2.

consume_value(*ctx: Context*, *opts: Mapping[str, Parameter]*) → tuple[Any, ParameterSource]

Intercept value consumption to detect explicit CLI usage.

Parameters
- **ctx** (*click.Context*) – Current Click context.
- **opts** (*Mapping[str, click.Parameter]*) – Parsed option tokens from the command line.

Returns
Tuple of (value, source) as returned by the parent implementation.

Return type
tuple[Any, click.core.ParameterSource]

mafw.tools.click_extensions.**_activation_script_path**(*shell: str*) → Path

Build the activation script path for a shell.

Parameters
shell (*str*) – Resolved shell name.

Returns
Target activation script path.

Return type
pathlib.Path

mafw.tools.click_extensions.**_completion_marker_block**(*shell: str*) → str

Build the activation block appended to the environment activation script.

This block executes all files in *$VIRTUAL_ENV/share/mafw/*_completion.<ext>*.

Parameters
shell (*str*) – Resolved shell name.

Returns
Marker block to append to the activation file.

Return type
str

mafw.tools.click_extensions.**_strip_completion_marker_block**(*content: str*) → str

Remove the completion block delimited by the MAFw markers.

Parameters
content (*str*) – Activation script content.

Returns
Content without the MAFw completion block.

Return type
str

mafw.tools.click_extensions.**_virtualenv_root**() → Path

Return the active virtual environment root.

Returns
Active virtual environment path.

Return type
pathlib.Path

Raises
click.ClickException – If `VIRTUAL_ENV` is missing.

mafw.tools.click_extensions.**check_ci_completion_guard**() → None

Check if the current environment is a CI environment.

If the `CI` environment variable is set, the function prints an informational message and exits the process with code 0.

mafw.tools.click_extensions.**completion_script_path**(*tool_name: str*, *shell: str*) → Path

Build the completion script path inside the active virtual environment.

Parameters

- **tool_name** (*str*) – The name of the tool (e.g., 'mafw', 'multiversion-doc', 'release-mgt').
- **shell** (*str*) – Resolved shell name.

Returns
Target completion script path.

Return type
pathlib.Path

mafw.tools.click_extensions.**completion_shell_from_env**(*shell_path: str | None*) → str

Resolve a completion shell from `$SHELL`.

The resolver supports the shells handled by Click completion generation: `bash`, `zsh`, and `fish`.

Parameters
shell_path (*str | None*) – The raw shell path as exposed by the environment.

Returns
The normalized shell name.

Return type
str

Raises
click.ClickException – If the shell cannot be determined or is unsupported.

mafw.tools.click_extensions.**completion_source_script**(*tool_name: str*, *shell: str*) → str

Generate the Click completion source script for the requested shell.

Parameters

- **tool_name** (*str*) – The name of the tool.
- **shell** (*str*) – Resolved shell name.

Returns
Completion script content.

Return type
str

mafw.tools.click_extensions.**install_completion**(*tool_name: str*, *shell: str*, *force: bool*, *script_path: Path*) → Path

Install Click completion for the requested shell.

Parameters

- **tool_name** (*str*) – The name of the tool.
- **shell** (*str*) – Resolved shell name.
- **force** (*bool*) – Reinstall even if completion is already loaded.
- **script_path** (*pathlib.Path*) – The target path for the completion script.

Returns
Installed completion script path.

Return type
pathlib.Path

mafw.tools.click_extensions.**is_script_already_installed**(*tool_name: str*, *shell: str*) → bool

Check if the completion script for the tool is already installed.

Parameters

- **tool_name** (*str*) – The name of the tool.
- **shell** (*str*) – Resolved shell name.

Returns
True if the completion script exists, False otherwise.

Return type
bool

mafw.tools.click_extensions.**resolve_completion_shell**(*shell: str*) → str

Normalize the requested completion shell.

Parameters
shell (*str*) – Shell selector from the CLI.

Returns
The resolved supported shell name.

Return type
str

Raises
click.ClickException – If the shell is unsupported.

mafw.tools.click_extensions.**uninstall_completion_files**(*tool_name: str*, *shell: str | None = None*) → None

Remove installed completion files and activation hooks.

Parameters

- **tool_name** (*str*) – The name of the tool.
- **shell** (*str | None*) – Optional shell selector. When omitted, all completion files for the tool are removed.

mafw.tools.db_tools

Database Tools

This module provides utility functions for working with database models using the Peewee ORM. It offers helper functions for creating key-value mappings, retrieving primary key information, and combining fields for composite keys or display purposes.

Functions

combine_fields(fields[, join_str])	Combine multiple database fields into a single concatenated string expression.
combine_pk(model[, alias_name, join_str])	Combine primary key fields of a database model into a single aliased field expression.
get_pk(model)	Retrieve the primary key fields of a database model.
make_kv(model, key, value)	Create a key-value mapping from a database model.

mafw.tools.db_tools.**combine_fields**(*fields: list[Field]*, *join_str: str = ' x '*) → Function

Combine multiple database fields into a single concatenated string expression.

This function creates an SQL CONCAT expression that combines multiple field values into a single string using the specified separator. It's particularly useful for creating composite keys or display strings from multiple fields.

Parameters

- **fields** (*list[peewee.Field]*) – List of field objects to be combined.
- **join_str** (*str*) – String to use as separator between fields. Defaults to ' x '.

Returns

A SQL CONCAT function expression combining the fields.

Return type

peewee.Function

mafw.tools.db_tools.**combine_pk**(*model: MAFwBaseModel | type[MAFwBaseModel]*, *alias_name: str = 'combo_pk'*, *join_str: str = ' x '*) → Alias

Combine primary key fields of a database model into a single aliased field expression.

This function retrieves the primary key fields from the given model using *get_pk()* and combines them into a single field expression. For models with a single primary key field, it simply aliases that field. For composite primary keys, it uses :func:combine_fields` to concatenate the fields with the specified separator.

Parameters

- **model** (*MAFwBaseModel | type[MAFwBaseModel]*) – The database model or model class to examine for primary key fields.
- **alias_name** (*str*) – The alias name to apply to the resulting field expression. Defaults to 'combo_pk'.
- **join_str** (*str*) – String to use as separator between primary key fields when combining. Defaults to ' x '.

Returns

An aliased field expression representing the combined primary key.

Return type

peewee.Alias

mafw.tools.db_tools.**get_pk**(*model: MAFwBaseModel | type[MAFwBaseModel]*) → list[Field]

Retrieve the primary key fields of a database model.

This function examines the primary key of the provided model and returns a list of field objects that constitute the primary key. For composite primary keys, it returns all constituent fields; for simple primary keys, it returns a list containing just the primary key field.

Parameters

model (*MAFwBaseModel | type[MAFwBaseModel]*) – The database model or model class to examine.

Returns

A list of field objects representing the primary key fields.

Return type
list[peewee.Field]

mafw.tools.db_tools.**make_kv**(*model:* MAFwBaseModel | *type[*MAFwBaseModel*]*, *key: Field*, *value: Field*) → dict[Any, Any]

Create a key-value mapping from a database model.

This function selects data from a given model using specified key and value fields, and returns a dictionary where keys are values from the key field and values are values from the value field.

Parameters

- **model** (*MAFwBaseModel | type[MAFwBaseModel]*) – The database model or model class to query.
- **key** (*peewee.Field*) – The field to use as dictionary keys.
- **value** (*peewee.Field*) – The field to use as dictionary values.

Returns
A dictionary mapping key field values to value field values.

Return type
dict[Any, Any]

Raises
AttributeError – If the model parameter doesn't have the required methods.

mafw.tools.file_tools

The module provides utilities for handling file, filename, hashing and so on.

Functions

file_checksum(filenames[, buf_size])	Generates the hexadecimal digest of a file or a list of files.
remove_widow_db_rows(models)	Removes widow rows from a database table.
verify_checksum(models)	Verifies the goodness of FileChecksumField.

mafw.tools.file_tools.**file_checksum**(*filenames: str | Path | Sequence[str | Path]*, *buf_size: int = 65536*) → str

Generates the hexadecimal digest of a file or a list of files.

The digest is calculated using the sha256 algorithm.

Parameters

- **filenames** (*str, Path, list*) – The filename or the list of filenames for digest calculations.
- **buf_size** (*int, Optional*) – The buffer size in bytes for reading the input files. Defaults to 64kB.

Returns
The hexadecimal digest.

Return type
str

mafw.tools.file_tools.**remove_widow_db_rows**(*models: list[Model | type[Model]] | Model | type[Model]*) → None

Removes widow rows from a database table.

According to MAFw architecture, the Database is mainly providing I/O support to the various processors.

This means that the processor retrieves a list of items from a database table for processing and subsequently updates a result table with the newly generated outputs.

Very often the input and output data are not stored directly in the database, but rather in files saved on the disc. In this case, the database is just providing a valid path where the input (or output) data can be found.

From this point of view, a **widow row** is a database entry in which the file referenced by the FilenameField has been deleted. A typical example is the following: the user wants a certain processor to regenerate a given result stored inside an output file. Instead of setting up a complex filter so that the processor receives only this element to process, the user can delete the actual output file and ask the processor to process all new items.

The provided `models` can be either a list or a single element, representing either an instance of a DB model or a model class. If a model class is provided, then a select over all its entries is performed.

The function will look at all fields of *FileNameField* and *FileNameListField* and check if it corresponds to an existing path or list of paths. If not, then the corresponding row is removed from the DB table.

Parameters
models (*list[Model | type(Model)] | Model | type(Model)*) – A list or a single Model instance or Model class for widow rows removal.

Raises
TypeError – if `models` is not of the right type.

mafw.tools.file_tools.**verify_checksum**(*models: list[Model | type[Model]] | Model | type[Model]*) → None

Verifies the goodness of FileChecksumField.

If in a model there is a FileChecksumField, this must be connected to a FileNameField or a FileNameListField in the same model. The goal of this function is to recalculate the checksum of the FileNameField / FileNameListField and compare it with the actual stored value. If the newly calculated value differs from the stored one, the corresponding row in the model will be removed, as it is no longer valid.

If a file is missing, then the checksum check is not performed, but the row is removed right away.

This function can be CPU and I/O intensive and last a lot, so use it with care, especially when dealing with long tables and large files.

Parameters
models (*list[Model | type(Model)] | Model | type(Model)*) – A list or a single Model instance or Model class for checksum verification.

Raises

- **TypeError** – if `models` is not of the right type.
- *mafw.mafw_errors.ModelError* – if the FileCheckSumField is referring to a FileNameField that does not exist.

mafw.tools.generics

Functions

deep_update(base_dict, update_dict[, copy_first])	Recursively updates a dictionary.

mafw.tools.generics.**deep_update**(*base_dict: dict[Any, Any]*, *update_dict: dict[Any, Any]*, *copy_first: bool = True*) → dict[Any, Any]

Recursively updates a dictionary.

If copy_first is set to False, then the base_dict is actually updated.

Parameters

- **base_dict** (*dict[Any, Any]*) – The dictionary to update.

- **update_dict** (*dict[Any, Any]*) – The dictionary containing the updated fields
- **copy_first** (*bool, Optional*) – Whether the base dictionary should be copied or updated. Defaults to True

Returns
The recursively updated dictionary

Return type
dict[Any, Any]

mafw.tools.loop_tuning

Mixin utilities for tuning loop execution parameters via steering files.

This module defines *LoopTuningMixin*, which exposes loop operational settings as *ActiveParameter* instances. This allows temporary tuning through steering files without modifying processor code.

Author
Bulgheroni Antonio (antonio.bulgheroni@ec.europa.eu)

Added in version v2.2.0.

Classes

LoopTuningMixin()	Provide steering-file access to loop operational parameters.

class mafw.tools.loop_tuning.**LoopTuningMixin**

Bases: object

Provide steering-file access to loop operational parameters.

This mixin is meant to be combined with *Processor* subclasses when tuning parallel for-loop performance. Once optimal values are found, users can remove the mixin and pass the values via the processor constructor.

static _validate_loop_type(*value: Any*) → *LoopType*

Validate loop type values supported by loop tuning.

Parameters
value (*Any*) – The loop type value to validate.

Returns
The validated loop type.

Return type
LoopType

Raises
ProcessorParameterError – If the loop type is not supported.

initialise_parameters() → None

Initialises parameters and applies loop-tuning overrides if provided.

The tuning parameters override constructor settings only when they are explicitly set in the steering file or via keyword arguments.

tuning_loop_type: *ActiveParameter*[str]

Loop type used by the processor.

tuning_max_workers: *ActiveParameter*[int]

Number of worker threads for parallel loops.

tuning_queue_batch_size: *ActiveParameter*[int]

Batch size for queue processing in the queue-based parallel loop.

tuning_queue_size: *ActiveParameter*[int]

Queue size used by *ParallelForLoopWithQueue*.

mafw.tools.pandas_tools

A collection of useful convenience functions for common pandas operations

Functions

group_and_aggregate_data_frame(data_frame, ...)	Utility function to perform dataframe groupby and aggregation.
slice_data_frame(input_data_frame[, ...])	Slice a data frame according to *slicing_dict*.

mafw.tools.pandas_tools.**group_and_aggregate_data_frame**(*data_frame: DataFrame*, *grouping_columns: Iterable[str]*, *aggregation_functions: Iterable[str | Callable[[Any], Any]]*) → DataFrame

Utility function to perform dataframe groupby and aggregation.

This function is a simple wrapper to perform group by and aggregation operations on a dataframe. The user must provide a list of columns to perform the group by on and a list of functions for the aggregation of the other columns.

The output dataframe will have the aggregated columns renamed as originalname_aggregationfunction.

> **Note**
>
> Only numeric columns (and columns that can be aggregated) will be included in the aggregation. String columns that are not used for grouping will be automatically excluded from aggregation.

Parameters

- **data_frame** (*pandas.DataFrame*) – The input data frame
- **grouping_columns** (*Iterable[str]*) – The list of columns to group by on.
- **aggregation_functions** (*Iterable[str | Callable[[Any], Any]]*) – The list of functions to be used for the aggregation of the not grouped columns.

Returns

The aggregated dataframe after the groupby operation.

Return type

pandas.DataFrame

mafw.tools.pandas_tools.**slice_data_frame**(*input_data_frame: DataFrame*, *slicing_dict: MutableMapping[str, Any] | None = None*, ***kwargs: Any*) → DataFrame

Slice a data frame according to *slicing_dict*.

The input data frame will be sliced using the items of the *slicing_dict* applying the loc operator in this way: `sliced = input_data_frame[(input_data_frame[key]==value)]`.

If the slicing_dict is empty, then the full input_data_frame is returned.

Instead of the slicing_dict, the user can also provide key and value pairs as keyword arguments.

```
slice_data_frame(data_frame, {'A':14})
```

is equivalent to

`slice_data_frame(data_frame, A=14)`.

If the user provides a keyword argument that also exists in the slicing_dict, then the keyword argument will update the slicing_dict.

No checks on the column name is done, should a label be missing, the loc method will raise a KeyError.

Parameters

- **input_data_frame** (*pd.DataFrame*) – The data frame to be sliced.
- **slicing_dict** (*dict, Optional*) – A dictionary with columns and values for the slicing. Defaults to None
- **kwargs** – Keyword arguments to be used instead of the slicing dictionary.

Returns
The sliced dataframe

Return type
pd.DataFrame

mafw.tools.parallel

Utilities for parallel execution in MAFw.

Added in version v2.1.0.

Functions

is_free_threading()	Check if Python is running in free-threading mode.

mafw.tools.parallel.**is_free_threading**() → bool

Check if Python is running in free-threading mode.

This function inspects the Python interpreter to determine whether it is running in free-threading mode (also known as "nogil" or "free-threaded"), which disables the Global Interpreter Lock (GIL).

Returns
True if the interpreter is running in free-threading mode, False otherwise.

Return type
bool

mafw.tools.regexp

Module implements some basic functions involving regular expressions.

Functions

extract_protocol(url)	Extract the protocol portion from a database connection URL.
normalize_sql_spaces(sql_string)	Normalize multiple consecutive spaces in SQL string to single spaces.
parse_processor_name(processor_string)	Parse a processor name string into name and replica identifier components.

mafw.tools.regexp.**extract_protocol**(*url: str*) → str | None

Extract the protocol portion from a database connection URL.

The extract_protocol function takes a database connection URL string as input and extracts the protocol portion (the part before "://"). This function is useful for identifying the database type from connection strings.

Parameters
url (*str*) – The url from which the protocol will be extracted.

Returns
The protocol or None, if the extraction failed

Return type
str | None

mafw.tools.regexp.**normalize_sql_spaces**(*sql_string: str*) → str

Normalize multiple consecutive spaces in SQL string to single spaces. Only handles spaces, preserves other whitespace characters.

Parameters
sql_string (*str*) – The SQL string for space normalization.

Returns
The normalized SQL command.

Return type
str

mafw.tools.regexp.**parse_processor_name**(*processor_string: str*) → tuple[str, str | None]

Parse a processor name string into name and replica identifier components.

Given a string in the form 'MyProcessorName#156a', returns a tuple ('MyProcessorName', '156a'). If the input string is 'MyProcessorName' only, then it returns ('MyProcessorName', None). If it gets 'MyProcessorName#', it returns ('MyProcessorName', None) but emits a warning informing of a possible malformed name.

The processor name must be a valid Python identifier (class name).

Parameters
processor_string (*str*) – The processor name string to parse.

Returns
A tuple of (name, replica_id) where replica_id can be None.

Return type
tuple[str, str | None]

Raises
UnknownProcessor – if the name part is empty or not a valid Python identifier

mafw.tools.shell_tools

Shell execution utilities.

This module provides unified functions for executing external commands via subprocess, with support for dry-runs, environment merging, and standardized output.

Module Attributes

CONSOLE	Shared console for standardized output.

Functions

run(command, *[, cwd, dry_run, env, quiet])	Execute a subprocess command and return a completed process object.
run_stdout(command, *[, dry_run, quiet])	Execute a command and return stripped standard output.

mafw.tools.shell_tools.**_to_command_line**(*command: str | list[str]*) → str

Convert a command into a printable shell-like string.

Parameters
command (*str | list[str]*) – Command expressed as a string or tokenized list.

Returns
Human-readable command line.

Return type
str

mafw.tools.shell_tools.**run**(*command: str | list[str]*, *, *cwd: Path | str | None = None*, *dry_run: bool = False*, *env: dict[str, str] | None = None*, *quiet: bool = False*, ***kwargs: Any*) → CompletedProcess[Any]

Execute a subprocess command and return a completed process object.

The command line is always printed with a standard prefix. In dry-run mode, execution is skipped and a successful synthetic `CompletedProcess` is returned.

Added in version 2.2: Add the *quiet* parameter to do not display the command being executed

Parameters

- **command** (*str | list[str]*) – Command to execute.
- **cwd** (*Path | str | None*) – Working directory for command execution.
- **dry_run** (*bool*) – If `True`, print the command without executing it.
- **env** (*dict[str, str] | None*) – Optional environment variables to merge with the current environment.
- **quiet** (*bool*) – Suppress the standard console banner when `True`.
- **kwargs** (*Any*) – Additional keyword arguments forwarded to `subprocess.run`.

Returns
Completed process produced by the command execution.

Return type
subprocess.CompletedProcess[Any]

mafw.tools.shell_tools.**run_stdout**(*command: str | list[str]*, *, *dry_run: bool = False*, *quiet: bool = False*, ***kwargs: Any*) → str

Execute a command and return stripped standard output.

Added in version 2.2: Add the *quiet* parameter to do not display the command being executed

Parameters

- **command** (*str | list[str]*) – Command to execute.
- **dry_run** (*bool*) – If `True`, do not execute the command.
- **quiet** (*bool*) – Suppress the standard console banner when `True`.
- **kwargs** (*Any*) – Additional keyword arguments forwarded to `run`.

Returns
Standard output stripped of leading and trailing whitespace.

Return type
str

mafw.tools.shell_tools.**CONSOLE = <console width=80 None>**

Shared console for standardized output.

mafw.tools.toml_tools

Tools for reading, writing, and validating MAFw TOML steering files.

Author
Bulgheroni Antonio

Description
Utilities to generate and load TOML steering files and related helpers.

Module Attributes

ENV_PATTERN	Regex matching supported environment variable expansion patterns.
ENV_ESCAPE_SENTINEL	Sentinel used to preserve escaped variable patterns.
MAX_ENV_RESOLUTION_PASSES	Maximum number of expansion passes applied to a single string.

Functions

dump_processor_parameters_to_toml(...)	Dumps a toml file with processor parameters.
generate_steering_file(output_file, processors)	Generates a steering file.
load_steering_file(steering_file[, ...])	Load a steering file for the execution framework.
load_steering_file_legacy(steering_file)	Load a steering file without any semantic validation.
path_encoder(obj)	Encoder for PathItem.
processor_validator(processors)	Validates that all items in the list are valid processor instances or classes.
resolve_config_env(config[, env])	Resolve environment variables in every string value of a configuration dictionary.
resolve_string(value, env)	Resolve environment variables within a string value.

Classes

PathItem(t, value, original, trivia)	TOML item representing a Path

class mafw.tools.toml_tools.**PathItem**(*t: StringType*, *value: str*, *original: str*, *trivia: Trivia*)

Bases: `String`

TOML item representing a Path

unwrap() → Path

Returns as pure python object (ppo)

mafw.tools.toml_tools.**_add_db_configuration**(*database_conf: dict[str, Any] | None = None*, *db_engine: str = 'sqlite'*, *doc: TOMLDocument | None = None*) → TOMLDocument

Add the DB configuration to the TOML document

The expected structure of the database_conf dictionary is one of these:

```
option1 = {
    'DBConfiguration': {
        'URL': 'sqlite:///:memory:',
        'parameters': {
            'sqlite': {
                'pragmas': {
                    'journal_mode': 'wal',
                    'cache_size': -64000,
                    'foreign_keys': 1,
                    'synchronous': 0,
                },
            },
        },
    }
}

option2 = {
    'URL': 'sqlite:///:memory:',
    'authentication': {
        'method': 'env',
        'username': 'POSTGRES_USER',
        'password': 'POSTGRES_PASS',
    },
    'parameters': {
        'postgresql': {
            'sslmode': 'require',
        },
    },
}
```

Parameters

- **database_conf** (*dict*) – A dictionary with the database configuration. See comments above. If None, then the default is used.
- **db_engine** (*str, Optional*) – The database engine. It is used only in case the provided database configuration is invalid to retrieve the default configuration. Defaults to sqlite.
- **doc** (*TOMLDocument, Optional*) – The TOML document to add the DB configuration. If None, one will be created.

Returns

The modified document.

Return type

TOMLDocument

Raises

UnknownDBEngine – if the *database_conf* is invalid and the db_engine is not yet implemented.

mafw.tools.toml_tools.**dump_processor_parameters_to_toml**(*processors: list[ProcessorClassProtocol] | ProcessorClassProtocol*, *output_file: Path | str*) → None

Dumps a toml file with processor parameters.

This helper function can be used when the parameters of one or many processors have to be dumped to a TOML file. For each Processor in the *processors* a table in the TOML file will be added with their parameters

is the shape of parameter name = value.

It must be noted that *processors* can be:

- a list of processor classes (list[type[Processor]])
- a list of processor instances (list[Processor]])
- one single processor class (type[Processor])
- one single processor instance (Processor)

What value of the parameters will be dumped?

Good question, have a look at this *explanation*.

param processors
One or more processors for which the parameters should be dumped.

type processors
list[ProcessorClassProtocol] | ProcessorClassProtocol

param output_file
The name of the output file for the dump.

type output_file
Path | str

raise KeyAlreadyPresent
if an attempt to add twice, the same processor is made.

raise TypeError
if the list contains items different from Processor classes and instances.

mafw.tools.toml_tools.**generate_steering_file**(*output_file: Path | str*, *processors: list[ProcessorClassProtocol] | ProcessorClassProtocol*, *database_conf: dict[str, Any] | None = None*, *db_engine: str = 'sqlite'*) → None

Generates a steering file.

Parameters

- **output_file** (*Path | str*) – The output filename where the steering file will be save.
- **processors** (*list[ProcessorClassProtocol] | ProcessorClassProtocol*) – The processors list for which the steering file will be generated.
- **database_conf** (*dict, Optional*) – The database configuration dictionary
- **db_engine** – A string representing the DB engine to be used. Possible values are: *sqlite*, *postgresql* and *mysql*.

Type
str

mafw.tools.toml_tools.**load_steering_file**(*steering_file: Path | str*, *validation_level: ValidationLevel | None = ValidationLevel.SEMANTIC*) → dict[str, Any]

Load a steering file for the execution framework.

Parameters

- **steering_file** (*Path, str*) – The path to the steering file.
- **validation_level** – Requested validation tier, or `None` to skip validation.

Returns
The configuration dictionary.

Return type
dict

Raises
mafw.mafw_errors.InvalidSteeringFile – if the validation level reports at least one issue.

mafw.tools.toml_tools.**load_steering_file_legacy**(*steering_file: Path | str*) → dict[str, Any]

Load a steering file without any semantic validation.

Parameters
steering_file (*Path, str*) – The path to the steering file.

Returns
The parsed steering dictionary.

Return type
dict

mafw.tools.toml_tools.**path_encoder**(*obj: Any*) → Item

Encoder for PathItem.

mafw.tools.toml_tools.**processor_validator**(*processors: list[*ProcessorClassProtocol*]*) → bool

Validates that all items in the list are valid processor instances or classes.

Parameters
processors (*list[*ProcessorClassProtocol*]*) – The list of items to be validated.

Returns
True if all items are valid.

Return type
bool

mafw.tools.toml_tools.**resolve_config_env**(*config: dict[str, Any]*, *env: Mapping[str, str] | None = None*) → dict[str, Any]

Resolve environment variables in every string value of a configuration dictionary.

Parameters

- **config** (*dict[str, Any]*) – Configuration dictionary to resolve.
- **env** (*Mapping[str, str] | None*) – Optional environment mapping; defaults to `os.environ`.

Returns
A new configuration dictionary with resolved values.

Return type
dict[str, Any]

Raises
ValueError – If a required variable is missing or expansion does not converge.

mafw.tools.toml_tools.**resolve_string**(*value: str*, *env: Mapping[str, str]*) → str

Resolve environment variables within a string value.

Parameters

- **value** (*str*) – Input string to resolve.
- **env** (*Mapping[str, str]*) – Environment mapping to use for substitution.

Returns
The resolved string.

Return type
str

Raises

ValueError – If a required variable is missing or expansion does not converge.

mafw.tools.toml_tools.**ENV_ESCAPE_SENTINEL = '__MAFW_ENV_ESCAPE__{'**

Sentinel used to preserve escaped variable patterns.

mafw.tools.toml_tools.**ENV_PATTERN = re.compile('\n \\$\\{ # opening ${\n (?P<name>[A-Za-z_][A-Za-z0-9_]*) # variable name\n (?:\n (?P<op>:-|:\\?) # operator (:- or :?)\n (?P<value>, re.VERBOSE)**

Regex matching supported environment variable expansion patterns.

mafw.tools.toml_tools.**MAX_ENV_RESOLUTION_PASSES = 10**

Maximum number of expansion passes applied to a single string.

14.1.21 mafw.ui

User interface modules.

In this subpackage, there are some implementations of possible user interfaces to connect the library to the execution framework.

Modules

abstract_user_interface	An abstract generic user interface.
console_user_interface	The console user interface.
rich_user_interface	The rich user interface.

mafw.ui.abstract_user_interface

An abstract generic user interface.

The module provides a generic user interface that can be implemented to allow MAFw to communicate with different user interfaces.

MAFw is designed to operate seamlessly without a user interface; however, users often appreciate the added benefit of communication between the process execution and themselves.

There are several different interfaces and different interface types (Command Line, Textual, Graphical…) and everyone has its own preferences. In order to be as generic as possible, MAFw is allowing for an abstract interface layer so that the user can either decide to use one of the few coming with MAFw or to implement the interface to their favorite interface.

Classes

UserInterfaceBase()	An abstract base user interface class that defines the interface for communicating with different user interfaces.
UserInterfaceMeta	A metaclass used for the creation of user interface

class mafw.ui.abstract_user_interface.**UserInterfaceBase**

Bases: `object`

An abstract base user interface class that defines the interface for communicating with different user interfaces.

This class provides a standardized way for MAFw to interact with various user interfaces including command-line, textual, and graphical interfaces. It defines the required methods that any concrete implementation must provide.

The interface allows for task management, progress reporting, status updates, and user interaction capabilities.

Added in version v1.0.

create_task(*task_name: str*, *task_description: str = ''*, *completed: int = 0*, *increment: int | None = None*, *total: int | None = None*, ***kwargs: Any*) → None

Create a new task.

Parameters

- **task_name** (*str*) – A unique identifier for the task. You cannot have more than 1 task with the same name in the whole execution. If you want to use the processor name, it is recommended to use the *unique_name*.
- **task_description** (*str, Optional*) – A short description for the task. Defaults to None.
- **completed** (*int, Optional*) – The amount of task already completed. Defaults to None.
- **increment** (*int, Optional*) – How much of the task has been done since last update. Defaults to None.
- **total** (*int, Optional*) – The total amount of task. Defaults to None.

display_progress_message(*message: str*, *i_item: int*, *n_item: int | None*, *frequency: float*) → None

Display a message during the process execution.

Parameters

- **message** (*str*) – The message to be displayed.
- **i_item** (*int*) – The current item enumerator.
- **n_item** (*int | None*) – The total number of items or None for an indeterminate progress (while loop).
- **frequency** (*float*) – How often (in percentage of n_item) to display the message.

enter_interactive_mode() → Generator[None, Any, None]

A context manager for entering interactive mode.

This method provides a way to temporarily switch to interactive mode, allowing for direct user interaction during processing. It should be used as a context manager with a `with` statement.

Added in version v2.0.0.

Returns
A context manager that yields control to the interactive section

Return type
Generator

prompt_question(*question: str*, ***kwargs: Any*) → Any

Prompt the user with a question and return their response.

This method should display a question to the user and wait for their input. The implementation may vary depending on the specific user interface being used.

Added in version v2.0.0.

Parameters

- **question** (*str*) – The question to be asked to the user.
- **kwargs** – Additional keyword arguments that might be used by specific implementations.

Returns
The user's response to the question.

Return type
Any

update_task(*task_name: str*, *completed: int = 0*, *increment: int | None = None*, *total: int | None = None*, ***kwargs: Any*) → None

Update an existing task.

Parameters

- **task_name** (*str*) – A unique identifier for the task. You cannot have more than 1 task with the same name in the whole execution. If you want to use the processor name, it is recommended to use the *replica_name*.
- **completed** (*int, Optional*) – The amount of task already completed. Defaults to None.
- **increment** (*int, Optional*) – How much of the task has been done since last update. Defaults to None.
- **total** (*int, Optional*) – The total amount of task. Defaults to None.

always_display_progress_message = 10

Threshold for displaying progress messages.

If the total number of events is below this value, then the progress message is always displayed, otherwise follow the standard update frequency.

name = 'base'

The name of the interface

class mafw.ui.abstract_user_interface.**UserInterfaceMeta**

Bases: type

A metaclass used for the creation of user interface

mafw.ui.console_user_interface

The console user interface.

The module provides a simple, still efficient user interface ideal for code execution of a headless system where it is not possible to observe the output in real-time. Nevertheless, important messages are logged via the logging library and thus it is also possible to save them to a file, if a proper logging handler is set up.

Classes

ConsoleInterface()	A console user interface.

class mafw.ui.console_user_interface.**ConsoleInterface**

Bases: *UserInterfaceBase*

A console user interface.

Ideal for execution in a headless environment.

Messages are sent via the logging system, so they can also be saved to a file is a logging handler is properly set up in the execution framework.

change_of_processor_status(*processor_name: str*, *old_status:* ProcessorStatus, *new_status:* ProcessorStatus) → None

Log a processor status change.

This method logs when a processor changes its status, using debug level logging.

Parameters

- **processor_name** (*str*) – The name of the processor whose status changed.
- **old_status** (ProcessorStatus) – The previous status of the processor.
- **new_status** (ProcessorStatus) – The new status of the processor.

create_task(*task_name: str*, *task_description: str = ''*, *completed: int = 0*, *increment: int | None = None*, *total: int | None = None*, ***kwargs: Any*) → None

Create a new task.

Parameters

- **task_name** (*str*) – A unique identifier for the task. You cannot have more than 1 task with the same name in the whole execution. If you want to use the processor name, it is recommended to use the *unique_name*.
- **task_description** (*str, Optional*) – A short description for the task. Defaults to None.
- **completed** (*int, Optional*) – The amount of task already completed. Defaults to None.
- **increment** (*int, Optional*) – How much of the task has been done since last update. Defaults to None.
- **total** (*int, Optional*) – The total amount of task. Defaults to None.

display_progress_message(*message: str*, *i_item: int*, *n_item: int | None*, *frequency: float*) → None

Display a progress message with item counter.

This method displays progress information at specified intervals based on the frequency parameter. It formats the message with a counter showing the current item and total items.

Parameters

- **message** (*str*) – The progress message to display.
- **i_item** (*int*) – The current item number (0-based index).
- **n_item** (*int | None*) – The total number of items, or None if unknown.
- **frequency** (*float*) – The frequency at which messages should be displayed (in seconds).

prompt_question(*question: str*, ***kwargs: Any*) → Any

Prompt the user with a question and return their input.

This method displays a question to the user and waits for their input. Additional keyword arguments can be passed through to the underlying input function.

Added in version v2.0.0.

Parameters

- **question** (*str*) – The question to display to the user.
- **kwargs** – Additional keyword arguments to pass to the input function.

Returns
The user's input.

Return type
Any

update_task(*task_name: str*, *completed: int | None = None*, *increment: int | None = None*, *total: int | None = None*, ***kwargs: Any*) → None

Update an existing task.

Parameters

- **task_name** (*str*) – A unique identifier for the task. You cannot have more than one task with the same name in the whole execution. If you want to use the processor name, it is recommended to use the *replica_name*.
- **completed** (*int, Optional*) – The amount of task already completed. Defaults to None.
- **increment** (*int, Optional*) – How much of the task has been done since last update. Defaults to None.
- **total** (*int, Optional*) – The total amount of task. Defaults to None.

name = 'console'

The name of the interface

mafw.ui.rich_user_interface

The rich user interface.

The module provides an implementation of the abstract user interface that takes advantage from the *rich* library. Progress bars and spinners are shown during the processor execution along with log messages including markup language. In order for this logging message to appear properly rendered, the logger should be connected to a RichHandler.

Classes

RichInterface([progress_kws])	Implementation of the interface for rich.

class mafw.ui.rich_user_interface.**RichInterface**(*progress_kws: dict[str, Any] | None = None*)

Bases: *UserInterfaceBase*

Implementation of the interface for rich.

Parameters

progress_kws (*dict, Optional*) – A dictionary of keywords passed to the *rich.Progress*. Defaults to None

change_of_processor_status(*processor_name: str*, *old_status:* ProcessorStatus, *new_status:* ProcessorStatus) → None

Display a message when a processor status changes.

This method logs a debug message indicating that a processor has changed its status. The message uses rich markup to highlight the processor name and new status.

Parameters

- **processor_name** (*str*) – The name of the processor whose status has changed.
- **old_status** (ProcessorStatus) – The previous status of the processor.
- **new_status** (ProcessorStatus) – The new status of the processor.

create_task(*task_name: str*, *task_description: str = ''*, *completed: int = 0*, *increment: int | None = None*, *total: int | None = None*, ***kwargs: Any*) → None

Create a new task.

Parameters

- **task_name** (*str*) – A unique identifier for the task. You cannot have more than 1 task with the same name in the whole execution. If you want to use the processor name, it is recommended to use the *unique_name*.

- **task_description** (*str, Optional*) – A short description for the task. Defaults to ‘’.
- **completed** (*int, Optional*) – The amount of task already completed. Defaults to 0.
- **increment** (*int, Optional*) – How much of the task has been done since last update. Defaults to None.
- **total** (*int, Optional*) – The total amount of task. Defaults to None.

display_progress_message(*message: str*, *i_item: int*, *n_item: int | None*, *frequency: float*) → None

Display a message during the process execution.

Parameters

- **message** (*str*) – The message to be displayed.
- **i_item** (*int*) – The current item enumerator.
- **n_item** (*int | None*) – The total number of items or None for an indeterminate progress (while loop).
- **frequency** (*float*) – How often (in percentage of n_item) to display the message.

enter_interactive_mode() → Generator[None, Any, None]

Context manager to temporarily switch to interactive mode.

This method temporarily stops the progress display to allow for interactive input while preserving the original transient state. After yielding control, it restores the progress display with appropriate spacing to avoid overwriting previous output.

Added in version v2.0.0.

Note

This method should be used within a `with` statement to ensure proper cleanup.

prompt_question(*question: str*, ***kwargs: Any*) → Any

Prompt the user with a question and return their response.

This method uses the rich library’s prompt functionality to ask the user a question. It supports various prompt types including confirmation, input, and choice prompts.

Added in version v2.0.0.

Parameters

- **question** (*str*) – The question to ask the user.
- **kwargs** – Additional arguments to pass to the prompt function.
- **prompt_type** – The type of prompt to use. Defaults to `rich.prompt.Confirm`.
- **console** – The console to use for the prompt. Defaults to None.
- **password** – Whether to hide input when prompting for passwords. Defaults to False.
- **choices** – List of valid choices for choice prompts. Defaults to None.
- **default** – Default value for prompts that support it. Defaults to None.
- **show_default** – Whether to show the default value. Defaults to True.
- **show_choices** – Whether to show available choices. Defaults to True.
- **case_sensitive** – Whether choices are case sensitive. Defaults to True.

Returns
The user's response based on the prompt type.

Return type
Any

update_task(*task_name: str*, *completed: int | None = None*, *increment: int | None = None*, *total: int | None = None*, ***kwargs: Any*) → None

Update an existing task.

Parameters

- **task_name** (*str*) – A unique identifier for the task. You cannot have more than one task with the same name in the whole execution. If you want to use the processor name, it is recommended to use the *replica_name*.
- **completed** (*int, Optional*) – The amount of task already completed. Defaults to 0.
- **increment** (*int, Optional*) – How much of the task has been done since last update. Defaults to None.
- **total** (*int, Optional*) – The total amount of task. Defaults to None.

name = 'rich'

The name of the interface

PYTHON MODULE INDEX

a

mafw.active, 152

d

mafw.db, 153
mafw.db.db_configurations, 154
mafw.db.db_connection, 154
mafw.db.db_filter, 155
mafw.db.db_model, 170
mafw.db.db_types, 176
mafw.db.db_wizard, 176
mafw.db.fields, 177
mafw.db.model_register, 179
mafw.db.std_tables, 182
mafw.db.trigger, 185
mafw.decorators, 190
mafw.devtools, 194
mafw.devtools.cli, 196
mafw.devtools.cli.cli, 196
mafw.devtools.cli.completion, 196
mafw.devtools.cli.dependencies, 196
mafw.devtools.cli.documentation, 197
mafw.devtools.cli.release, 198
mafw.devtools.cli.toolchain, 198
mafw.devtools.cli.toolchain.bootstrap_cmd, 199
mafw.devtools.cli.toolchain.check_cmd, 199
mafw.devtools.cli.toolchain.list_cmd, 199
mafw.devtools.cli.toolchain.update_cmd, 199
mafw.devtools.cli.toolchain.verify_cmd, 199
mafw.devtools.dependencies, 200
mafw.devtools.dependencies.audit, 208
mafw.devtools.dependencies.compare, 209
mafw.devtools.dependencies.compile, 212
mafw.devtools.dependencies.freeze, 215
mafw.devtools.dependencies.verify, 220
mafw.devtools.documentation, 220
mafw.devtools.documentation.builder, 233
mafw.devtools.documentation.requirements, 239
mafw.devtools.documentation.server, 240
mafw.devtools.documentation.versions, 243
mafw.devtools.git, 246
mafw.devtools.gitlab, 248
mafw.devtools.gitlab.api, 253
mafw.devtools.gitlab.docs_registry, 254
mafw.devtools.gitlab.registry, 257
mafw.devtools.release, 262
mafw.devtools.release.changelog, 269
mafw.devtools.release.checks, 270
mafw.devtools.release.notes, 271
mafw.devtools.release.versioning, 272
mafw.devtools.toolchain, 277
mafw.devtools.toolchain.base, 284
mafw.devtools.toolchain.default_registry, 291
mafw.devtools.toolchain.filtering, 292
mafw.devtools.toolchain.precommit_modifier, 293
mafw.devtools.toolchain.pypi, 294
mafw.devtools.toolchain.pyproject_modifier, 295
mafw.devtools.toolchain.registry, 298
mafw.devtools.toolchain.tools, 299
mafw.devtools.toolchain.tools.git_cliff, 307
mafw.devtools.toolchain.tools.hatch, 308
mafw.devtools.toolchain.tools.mypy, 309
mafw.devtools.toolchain.tools.pip_audit, 310
mafw.devtools.toolchain.tools.precommit, 311
mafw.devtools.toolchain.tools.pytest, 312
mafw.devtools.toolchain.tools.pyupgrade, 314
mafw.devtools.toolchain.tools.ruff, 316
mafw.devtools.toolchain.tools.sphinx, 318
mafw.devtools.toolchain.tools.uv, 320

e

mafw.enumerators, 321
mafw.examples, 323
mafw.examples.db_processors, 324
mafw.examples.importer_example, 326
mafw.examples.loop_modifier, 329
mafw.examples.multi_primary, 335
mafw.examples.processor_list, 335
mafw.examples.sum_processor, 336

h

mafw.hookspecs, 337

l

mafw.lazy_import, 337

m

mafw, 151
mafw.mafw_errors, 340
mafw.models, 344
mafw.models.filter_schema, 345
mafw.models.loop_payloads, 345
mafw.models.parameter_schema, 346
mafw.models.processor_schema, 347

p

mafw.plugin_manager, 347
mafw.plugins, 351
mafw.processor, 351
mafw.processor.base, 368
mafw.processor.meta, 379
mafw.processor.parameters, 380
mafw.processor.processor_list, 385
mafw.processor.utils, 387
mafw.processor_library, 387
mafw.processor_library.abstract_plotter, 387
mafw.processor_library.db_init, 392
mafw.processor_library.importer, 399
mafw.processor_library.sns_plotter, 404

r

mafw.runner, 414

s

mafw.scripts, 416
mafw.scripts.mafw_exe, 416
mafw.steering, 426
mafw.steering.builder, 426
mafw.steering.models, 429
mafw.steering.serializer, 434
mafw.steering.validation, 434
mafw.steering_gui, 434
mafw.steering_gui.actions, 435
mafw.steering_gui.app, 435
mafw.steering_gui.controllers, 436
mafw.steering_gui.controllers.steering_controller, 436
mafw.steering_gui.dialogs, 442
mafw.steering_gui.dialogs.about_dialog, 443
mafw.steering_gui.dialogs.add_field_filter_dialog, 443
mafw.steering_gui.dialogs.add_model_dialog, 444
mafw.steering_gui.dialogs.add_processor_dialog, 444
mafw.steering_gui.dialogs.steering_text_editor, 444
mafw.steering_gui.dialogs.validation_issues_dialog, 445
mafw.steering_gui.main_window, 445
mafw.steering_gui.models, 449
mafw.steering_gui.models.filter_tree_model, 450
mafw.steering_gui.models.pipeline, 452
mafw.steering_gui.models.processor_pipeline_model, 453
mafw.steering_gui.models.steering_tree_model, 454
mafw.steering_gui.threading, 454
mafw.steering_gui.threading.workers, 455
mafw.steering_gui.utils, 456
mafw.steering_gui.utils.constants, 456
mafw.steering_gui.utils.exception_handling, 456
mafw.steering_gui.utils.gui_helpers, 458
mafw.steering_gui.views, 458
mafw.steering_gui.views.database_editor, 461
mafw.steering_gui.views.filter_editors, 464
mafw.steering_gui.views.filter_manager, 468
mafw.steering_gui.views.filter_tree, 469
mafw.steering_gui.views.globals_editor, 470
mafw.steering_gui.views.group_editor, 470
mafw.steering_gui.views.meta_editor, 471
mafw.steering_gui.views.pipeline_editor, 471
mafw.steering_gui.views.processor_editor, 472
mafw.steering_gui.views.processor_parameter_editor, 472
mafw.steering_gui.views.steering_tree, 473
mafw.steering_gui.views.ui_editor, 474

t

mafw.timer, 474
mafw.tools, 476
mafw.tools.click_extensions, 476
mafw.tools.db_tools, 480
mafw.tools.file_tools, 482
mafw.tools.generics, 483
mafw.tools.loop_tuning, 484
mafw.tools.pandas_tools, 485
mafw.tools.parallel, 486
mafw.tools.regexp, 486
mafw.tools.shell_tools, 487
mafw.tools.toml_tools, 489

u

mafw.ui, 493
mafw.ui.abstract_user_interface, 493
mafw.ui.console_user_interface, 495
mafw.ui.rich_user_interface, 497

INDEX

Symbols

_ACTIVE_COLUMN_LABEL (*in module mafw.steering_gui.views.processor_parameter_editor*), 473
_COLUMNS (*in module mafw.steering_gui.views.processor_parameter_editor*), 473
_CONTENT_TYPE_MAP (*in module mafw.devtools.gitlab.registry*), 262
_DEFAULT_DESCRIPTION (*in module mafw.steering_gui.views.group_editor*), 471
_DEFAULT_GROUP_NAME (*in module mafw.steering_gui.views.group_editor*), 471
_DEFAULT_SAVE_NAME (*in module mafw.steering_gui.main_window*), 448
_DisplayCallback (*in module mafw.steering_gui.utils.exception_handling*), 458
_F (*class in mafw.processor.parameters*), 383
_FIELD_TYPE_STRINGS (*in module mafw.steering.models*), 433
_FROZEN_OPERATORS (*in module mafw.devtools.dependencies.freeze*), 219
_GroupTableView (*class in mafw.steering_gui.views.group_editor*), 470
_HATCH_UV_SECTION (*in module mafw.devtools.toolchain.tools.uv*), 321
_LOWER_BOUND_RE (*in module mafw.devtools.toolchain.pyproject_modifier*), 298
_MAFW_VERSION_RE (*in module mafw.devtools.gitlab.registry*), 262
_MAX_REV_LENGTH (*in module mafw.devtools.toolchain.precommit_modifier*), 294
_NO_VALUE_OPS (*in module mafw.steering_gui.views.filter_editors*), 468
_OPERATOR_TOOLTIPS (*in module mafw.steering_gui.views.filter_editors*), 468
_PEP503_NORMALIZE_RE (*in module mafw.devtools.toolchain.pyproject_modifier*), 298
_PRECOMMIT_REPO_URL (*in module mafw.devtools.toolchain.tools.pyupgrade*), 316
_PRECOMMIT_REPO_URL (*in module mafw.devtools.toolchain.tools.ruff*), 318
_PYLOCK_RE (*in module mafw.devtools.gitlab.registry*), 262
_PYPI_JSON_URL (*in module mafw.devtools.toolchain.pypi*), 295
_PYPROJECT_PATH (*in module mafw.devtools.toolchain.tools.uv*), 321
_ParameterRegistry (*in module mafw.processor.parameters*), 384
_ParameterRoles (*class in mafw.steering_gui.views.processor_parameter_editor*), 473
_ParameterRow (*class in mafw.steering_gui.views.processor_parameter_editor*), 473
_PipelineTableView (*class in mafw.steering_gui.views.pipeline_editor*), 472
_REQUEST_TIMEOUT_S (*in module mafw.devtools.toolchain.pypi*), 295
_RESOURCE_DIR (*in module mafw.steering_gui.models.filter_tree_model*), 452
_SectionNode (*class in mafw.steering_gui.models.steering_tree_model*), 454
_VALIDATION_DEBOUNCE_MS (*in module mafw.steering_gui.dialogs.steering_text_editor*), 445
_WARNING_ICON_PATH (*in module mafw.steering_gui.models.filter_tree_model*), 452
_activation_script_path() (*in module mafw.tools.click_extensions*), 478
_add_db_configuration() (*in module mafw.tools.toml_tools*), 489
_adjust_column_widths()

(*mafw.steering_gui.dialogs.validation_issues_dialog.ValidationIssuesDialog method*), 445
_apply_super_call_wrappers() (*mafw.processor.Processor class method*), 355
_apply_super_call_wrappers() (*mafw.processor.base.Processor class method*), 369
_as_db_module_result() (*in module mafw.plugin_manager*), 350
_as_processor_result() (*in module mafw.plugin_manager*), 350
_as_ui_result() (*in module mafw.plugin_manager*), 350
_assemble_connection_source() (*mafw.steering_gui.controllers.steering_controller.SteeringController method*), 437
_attributes_valid() (*mafw.processor_library.sns_plotter.FromDatasetDataRetriever method*), 407
_attributes_valid() (*mafw.processor_library.sns_plotter.SQLPdDataRetriever method*), 413
_auto_bind() (*mafw.db.db_filter.ModelFilter method*), 165
_build_logical_node_from_ast() (*mafw.db.db_filter.ModelFilter method*), 166
_build_loop_item() (*mafw.processor.Processor method*), 355
_build_loop_item() (*mafw.processor.base.Processor method*), 369
_build_loop_result() (*mafw.processor.Processor method*), 356
_build_loop_result() (*mafw.processor.base.Processor method*), 369
_call_with_optional_payload() (*mafw.processor.Processor method*), 356
_call_with_optional_payload() (*mafw.processor.base.Processor method*), 370
_check_method_overload() (*mafw.processor.Processor method*), 356
_check_method_overload() (*mafw.processor.base.Processor method*), 370
_classify_changelog_subsection() (*in module mafw.devtools.release.changelog*), 269
_clean_sql() (*mafw.db.trigger.PostgreSQLDialect method*), 185
_collect_condition_keys() (*mafw.steering_gui.models.filter_tree_model.FilterTreeModel method*), 451
_collect_logic_keys() (*mafw.steering_gui.models.filter_tree_model.FilterTreeModel method*), 451
_compile_conditions() (*mafw.db.trigger.Trigger method*), 187
_compile_query() (*mafw.db.trigger.Trigger method*), 187
_compile_sql_statement() (*mafw.db.trigger.Trigger method*), 187
_compile_when_conditions() (*mafw.db.trigger.Trigger method*), 187
_completion_marker_block() (*in module mafw.tools.click_extensions*), 478
_compute_default_max_workers() (*mafw.processor.Processor static method*), 355
_compute_default_max_workers() (*mafw.processor.base.Processor static method*), 369
_config (*mafw.processor.Processor attribute*), 363
_config (*mafw.processor.base.Processor attribute*), 377
_configuration_file (*mafw.processor_library.importer.FilenameParser attribute*), 402
_confirm_discard() (*mafw.steering_gui.main_window.MainWindow method*), 446
_copy_parent_parameter_definitions() (*in module mafw.processor.parameters*), 383
_create_actions() (*mafw.steering_gui.main_window.MainWindow method*), 446
_create_central_widget() (*mafw.steering_gui.main_window.MainWindow method*), 446
_create_condition_node_from_value() (*mafw.db.db_filter.ModelFilter static method*), 165
_create_group() (*mafw.steering_gui.main_window.MainWindow method*), 446
_create_menus() (*mafw.steering_gui.main_window.MainWindow method*), 446
_create_status_bar() (*mafw.steering_gui.main_window.MainWindow method*), 446
_create_value_widget() (*in module mafw.steering_gui.views.filter_editors*), 468
_current_position() (*mafw.db.db_filter.ExprParser method*), 160
_delayed_status_message() (*mafw.plugin_manager.MAFwPluginManager method*), 349
_detect_backend_from_url() (*mafw.steering_gui.views.DatabaseEditor method*), 458
_detect_backend_from_url() (*mafw.steering_gui.views.database_editor.DatabaseEditor method*), 462
_detect_command_name()

(*mafw.scripts.mafw_exe.MAFwGroup method*), 425
_devtools_checked (*in module mafw.devtools*), 195
_did_call_super() (*mafw.processor.Processor method*), 356
_did_call_super() (*mafw.processor.base.Processor method*), 370
_display_section() (*mafw.steering_gui.main_window.MainWindow method*), 446
_element_dict (*mafw.processor_library.importer.FilenameParser attribute*), 402
_ensure_parameter_definitions() (*in module mafw.processor.parameters*), 384
_ensure_pipx_available() (*mafw.devtools.toolchain.HostTool method*), 277
_ensure_pipx_available() (*mafw.devtools.toolchain.base.HostTool method*), 285
_evaluate_logic_ast() (*mafw.db.db_filter.ModelFilter method*), 166
_execute_for_loop() (*mafw.processor.Processor method*), 356
_execute_for_loop() (*mafw.processor.base.Processor method*), 370
_execute_single() (*mafw.processor.Processor method*), 356
_execute_single() (*mafw.processor.base.Processor method*), 370
_execute_while_loop() (*mafw.processor.Processor method*), 356
_execute_while_loop() (*mafw.processor.base.Processor method*), 370
_exit_app() (*mafw.steering_gui.main_window.MainWindow method*), 446
_expand_processors_to_run() (*mafw.runner.MAFwApplication method*), 415
_extract_package_name() (*in module mafw.devtools.toolchain.pyproject_modifier*), 297
_extract_raw_rev() (*in module mafw.devtools.toolchain.precommit_modifier*), 294
_extract_specifier_from_requirement() (*in module mafw.devtools.toolchain.tools.uv*), 321
_field_type_from_peewee() (*in module mafw.steering.models*), 433
_filename (*mafw.processor_library.importer.FilenameParser attribute*), 402
_filename_parser (*mafw.processor_library.importer.Importer attribute*), 404
_finalize_connection_test() (*mafw.steering_gui.main_window.MainWindow method*), 446
_first_unused_model_field() (*mafw.steering_gui.views.filter_tree.FilterTreeView method*), 469
_format_expected() (*in module mafw.db.db_filter*), 168
_generate_group_name() (*mafw.steering_gui.main_window.MainWindow method*), 446
_generate_next_value_() (*mafw.db.trigger.TriggerAction static method*), 189
_generate_next_value_() (*mafw.db.trigger.TriggerWhen static method*), 190
_generate_next_value_() (*mafw.devtools.documentation.ServerStatusEnum static method*), 221
_generate_next_value_() (*mafw.devtools.documentation.server.ServerStatusEnum static method*), 241
_generate_next_value_() (*mafw.enumerators.LoopType static method*), 322
_generate_next_value_() (*mafw.enumerators.ProcessorStatus static method*), 323
_generate_next_value_() (*mafw.steering.models.FilterKind static method*), 431
_generate_next_value_() (*mafw.steering.models.ParameterSchemaStatus static method*), 432
_generate_next_value_() (*mafw.steering.models.ParameterSource static method*), 432
_generate_next_value_() (*mafw.steering.models.ProcessorSchemaStatus static method*), 433
_get_dialect() (*mafw.db.trigger.Trigger method*), 187
_get_highest_python_version() (*mafw.devtools.toolchain.ProjectTool method*), 279
_get_highest_python_version() (*mafw.devtools.toolchain.base.ProjectTool method*), 287
_get_modifier() (*mafw.devtools.toolchain.ProjectTool method*), 279
_get_modifier() (*mafw.devtools.toolchain.base.ProjectTool method*), 287
_get_pipx_version() (*mafw.devtools.toolchain.HostTool method*), 277
_get_pipx_version() (*mafw.devtools.toolchain.base.HostTool method*), 285

_get_precommit_version() (*mafw.devtools.toolchain.tools.RuffTool method*), 304
_get_precommit_version() (*mafw.devtools.toolchain.tools.pyupgrade.PyUpgradeTool method*), 314
_get_precommit_version() (*mafw.devtools.toolchain.tools.ruff.RuffTool method*), 317
_get_value_type() (*mafw.processor_library.importer.FilenameImporter class method*), 400
_get_widget_value() (*in module mafw.steering_gui.views.filter_editors*), 468
_has_logic() (*mafw.steering_gui.models.filter_tree_model.FilterTreeModel method*), 451
_has_logic() (*mafw.steering_gui.views.filter_tree.FilterTreeView method*), 469
_ids (*mafw.processor.Processor attribute*), 363
_ids (*mafw.processor.base.Processor attribute*), 377
_in_thread_context() (*mafw.processor.Processor method*), 356
_in_thread_context() (*mafw.processor.base.Processor method*), 370
_infer_content_type() (*in module mafw.devtools.gitlab.registry*), 258
_is_dict_annotation() (*mafw.processor.ActiveParameter method*), 353
_is_dict_annotation() (*mafw.processor.parameters.ActiveParameter method*), 381
_is_list_annotation() (*mafw.processor.ActiveParameter method*), 353
_is_list_annotation() (*mafw.processor.parameters.ActiveParameter method*), 381
_item (*mafw.processor.Processor attribute*), 363
_item (*mafw.processor.base.Processor attribute*), 377
_load() (*mafw.lazy_import.LazyImportPlugin method*), 338
_load_default_python_version() (*in module mafw.devtools.documentation.requirements*), 239
_load_file() (*mafw.steering_gui.main_window.MainWindow method*), 446
_load_parameter_configuration() (*mafw.processor.Processor method*), 357
_load_parameter_configuration() (*mafw.processor.base.Processor method*), 370
_load_supported_python_versions() (*in module mafw.devtools.documentation*), 221
_load_supported_python_versions() (*in module mafw.devtools.documentation.requirements*), 239
_lock (*mafw.examples.loop_modifier.FindPrimeNumberInRangeParallel attribute*), 332
_lock (*mafw.examples.loop_modifier.FindPrimeNumberInRangeTunable attribute*), 334
_looping_status (*mafw.processor.Processor attribute*), 363
_looping_status (*mafw.processor.base.Processor attribute*), 377
_mark_clean() (*mafw.steering_gui.controllers.steering_controller.SteeringController method*), 437
_mark_super_call() (*mafw.processor.Processor method*), 357
_mark_super_call() (*mafw.processor.base.Processor method*), 371
_matches_container() (*mafw.processor.ActiveParameter static method*), 353
_matches_container() (*mafw.processor.parameters.ActiveParameter static method*), 381
_methods_to_be_checked_for_super (*mafw.processor.Processor attribute*), 363
_methods_to_be_checked_for_super (*mafw.processor.base.Processor attribute*), 377
_methods_to_be_checked_for_super (*mafw.processor_library.sns_plotter.SNSPlotter attribute*), 412
_new_file() (*mafw.steering_gui.main_window.MainWindow method*), 446
_node_to_sql() (*mafw.db.trigger.Trigger method*), 187
_normalize_config() (*mafw.steering_gui.views.DatabaseEditor method*), 459
_normalize_config() (*mafw.steering_gui.views.database_editor.DatabaseEditor method*), 462
_normalize_field_type() (*in module mafw.steering.models*), 433
_normalize_logical_op() (*in module mafw.steering.serializer*), 434
_normalize_name() (*in module mafw.devtools.toolchain.pyproject_modifier*), 297
_normalize_schema_default() (*mafw.steering_gui.controllers.steering_controller.SteeringController static method*), 437
_normalize_text() (*mafw.steering_gui.main_window.MainWindow static method*), 446
_on_add_group_requested() (*mafw.steering_gui.main_window.MainWindow method*), 446
_on_add_processor_requested() (*mafw.steering_gui.main_window.MainWindow method*), 446
_on_conditional_filter_changed()

(*mafw.steering_gui.views.filter_manager.FilterManager method*), 469
_on_connection_test_failure() (*mafw.steering_gui.main_window.MainWindow method*), 446
_on_connection_test_success() (*mafw.steering_gui.main_window.MainWindow method*), 446
_on_database_configuration_changed() (*mafw.steering_gui.main_window.MainWindow method*), 446
_on_dirty_changed() (*mafw.steering_gui.main_window.MainWindow method*), 447
_on_group_add_group_requested() (*mafw.steering_gui.main_window.MainWindow method*), 447
_on_group_add_processor_requested() (*mafw.steering_gui.main_window.MainWindow method*), 447
_on_group_description_changed() (*mafw.steering_gui.main_window.MainWindow method*), 447
_on_group_move_requested() (*mafw.steering_gui.main_window.MainWindow method*), 447
_on_group_name_changed() (*mafw.steering_gui.main_window.MainWindow method*), 447
_on_group_remove_requested() (*mafw.steering_gui.main_window.MainWindow method*), 447
_on_pipeline_edit_requested() (*mafw.steering_gui.main_window.MainWindow method*), 447
_on_pipeline_error() (*mafw.steering_gui.main_window.MainWindow method*), 447
_on_pipeline_move_requested() (*mafw.steering_gui.main_window.MainWindow method*), 447
_on_pipeline_ready() (*mafw.steering_gui.main_window.MainWindow method*), 447
_on_pipeline_remove_requested() (*mafw.steering_gui.main_window.MainWindow method*), 447
_on_pipeline_selection_changed() (*mafw.steering_gui.main_window.MainWindow method*), 447
_on_plugin_load_failure() (*mafw.steering_gui.main_window.MainWindow method*), 447
_on_plugins_loaded() (*mafw.steering_gui.main_window.MainWindow method*), 447
_on_selection_changed() (*mafw.steering_gui.views.SteeringTreeView method*), 460
_on_selection_changed() (*mafw.steering_gui.views.steering_tree.SteeringTreeView method*), 474
_on_test_connection_requested() (*mafw.steering_gui.main_window.MainWindow method*), 447
_on_ui_interface_changed() (*mafw.steering_gui.main_window.MainWindow method*), 447
_on_validation_level_changed() (*mafw.steering_gui.main_window.MainWindow method*), 447
_open_file() (*mafw.steering_gui.main_window.MainWindow method*), 447
_open_steering_text_editor() (*mafw.steering_gui.main_window.MainWindow method*), 447
_operators_for_type() (*in module mafw.steering_gui.views.filter_editors*), 468
_orig_config (*mafw.processor.Processor attribute*), 363
_orig_config (*mafw.processor.base.Processor attribute*), 377
_override_defaults() (*mafw.processor.Processor method*), 357
_override_defaults() (*mafw.processor.base.Processor method*), 371
_overrule_kws_parameters() (*mafw.processor.Processor method*), 357
_overrule_kws_parameters() (*mafw.processor.base.Processor method*), 371
_parameter_registered (*mafw.processor.Processor attribute*), 363
_parameter_registered (*mafw.processor.base.Processor attribute*), 377
_parse_major_minor() (*in module mafw.devtools.release.versioning*), 274
_parse_resolution_failure() (*mafw.devtools.toolchain.ProjectTool static method*), 279
_parse_resolution_failure() (*mafw.devtools.toolchain.base.ProjectTool static method*), 287
_parser_configuration() (*mafw.processor_library.importer.FilenameParser method*), 402
_payload_annotation_matches() (*mafw.processor.Processor method*), 357
_payload_annotation_matches() (*mafw.processor.base.Processor method*), 371
_perform_save() (*mafw.steering_gui.main_window.MainWindow method*), 447

_perform_save_as() (*mafw.steering_gui.main_window.MainWindow method*), 448
_populate_from_model() (*mafw.steering_gui.main_window.MainWindow method*), 448
_post_load() (*mafw.lazy_import.LazyImportPlugin method*), 338
_post_load() (*mafw.lazy_import.LazyImportProcessor method*), 338
_post_load() (*mafw.lazy_import.LazyImportUserInterface method*), 339
_process_for_loop() (*mafw.processor.Processor method*), 358
_process_for_loop() (*mafw.processor.base.Processor method*), 371
_process_parallel_for_loop() (*mafw.processor.Processor method*), 358
_process_parallel_for_loop() (*mafw.processor.base.Processor method*), 371
_process_parallel_for_loop_with_queue() (*mafw.processor.Processor method*), 358
_process_parallel_for_loop_with_queue() (*mafw.processor.base.Processor method*), 372
_processor_parameters (*mafw.processor.Processor attribute*), 363
_processor_parameters (*mafw.processor.base.Processor attribute*), 377
_read_devtools_deps() (*in module mafw.devtools*), 195
_recreate_environment() (*mafw.devtools.toolchain.ProjectTool method*), 280
_recreate_environment() (*mafw.devtools.toolchain.base.ProjectTool method*), 288
_redo() (*mafw.steering_gui.main_window.MainWindow method*), 448
_refresh_database_editor() (*mafw.steering_gui.main_window.MainWindow method*), 448
_refresh_globals_editor() (*mafw.steering_gui.main_window.MainWindow method*), 448
_refresh_metadata_summary() (*mafw.steering_gui.main_window.MainWindow method*), 448
_refresh_pipeline_after_change() (*mafw.steering_gui.main_window.MainWindow method*), 448
_refresh_title_from_controller() (*mafw.steering_gui.main_window.MainWindow method*), 448
_refresh_ui_editor() (*mafw.steering_gui.main_window.MainWindow method*), 448
_register_parameters() (*mafw.processor.Processor method*), 358
_register_parameters() (*mafw.processor.base.Processor method*), 372
_remove_orphan_files() (*mafw.processor.Processor method*), 358
_remove_orphan_files() (*mafw.processor.base.Processor method*), 372
_replace_rev_after_repo() (*in module mafw.devtools.toolchain.precommit_modifier*), 294
_reset_parameters() (*mafw.processor.Processor method*), 358
_reset_parameters() (*mafw.processor.base.Processor method*), 372
_reset_super_call_flag() (*mafw.processor.Processor method*), 359
_reset_super_call_flag() (*mafw.processor.base.Processor method*), 372
_resolve_parameter_annotation() (*mafw.processor.ActiveParameter method*), 353
_resolve_parameter_annotation() (*mafw.processor.parameters.ActiveParameter method*), 382
_resolve_section() (*mafw.devtools.toolchain.pyproject_modifier.PyprojectModifier method*), 296
_restore_window_geometry() (*mafw.steering_gui.main_window.MainWindow method*), 448
_revert() (*mafw.devtools.toolchain.ProjectTool method*), 280
_revert() (*mafw.devtools.toolchain.base.ProjectTool method*), 288
_revert_all() (*mafw.devtools.toolchain.tools.RuffTool method*), 304
_revert_all() (*mafw.devtools.toolchain.tools.pyupgrade.PyUpgradeTo method*), 315
_revert_all() (*mafw.devtools.toolchain.tools.ruff.RuffTool method*), 317
_revert_precommit() (*mafw.devtools.toolchain.tools.RuffTool method*), 304
_revert_precommit() (*mafw.devtools.toolchain.tools.pyupgrade.PyUpgradeTool method*), 315
_revert_precommit() (*mafw.devtools.toolchain.tools.ruff.RuffTool method*), 317
_save_file() (*mafw.steering_gui.main_window.MainWindow method*), 448

_save_file_as() (*mafw.steering_gui.main_window.MainWindow method*), 448
_save_window_geometry() (*mafw.steering_gui.main_window.MainWindow method*), 448
_schema_default_value() (*mafw.processor.ActiveParameter method*), 353
_schema_default_value() (*mafw.processor.parameters.ActiveParameter method*), 382
_serialize_condition() (*in module mafw.steering.serializer*), 434
_set_test_button_enabled() (*mafw.steering_gui.main_window.MainWindow method*), 448
_set_widget_value() (*in module mafw.steering_gui.views.filter_editors*), 468
_setup_model_data() (*mafw.steering_gui.models.filter_tree_model.FilterTreeModel method*), 451
_setup_ui() (*mafw.steering_gui.dialogs.AboutDialog method*), 442
_setup_ui() (*mafw.steering_gui.dialogs.about_dialog.AboutDialog method*), 443
_show_about() (*mafw.steering_gui.main_window.MainWindow method*), 448
_split_section_path() (*mafw.devtools.toolchain.pyproject_modifier.PyprojectModifier static method*), 296
_standard_table_models() (*in module mafw.processor_library.db_init*), 399
_start_connection_test_worker() (*mafw.steering_gui.main_window.MainWindow method*), 448
_start_plugin_loading() (*mafw.steering_gui.main_window.MainWindow method*), 448
_strip_completion_marker_block() (*in module mafw.tools.click_extensions*), 478
_super_call_flags (*mafw.processor.Processor attribute*), 363
_super_call_flags (*mafw.processor.base.Processor attribute*), 377
_thread_exception_handler (*in module mafw.steering_gui.utils.exception_handling*), 458
_thread_local (*mafw.processor.Processor attribute*), 364
_thread_local (*mafw.processor.base.Processor attribute*), 377
_thread_loop_context() (*mafw.processor.Processor method*), 359
_thread_loop_context() (*mafw.processor.base.Processor method*), 373
_to_command_line() (*in module mafw.tools.shell_tools*), 488
_tokenize_with_positions() (*in module mafw.db.db_filter*), 168
_undo() (*mafw.steering_gui.main_window.MainWindow method*), 448
_unique_processor_name() (*mafw.steering_gui.controllers.steering_controller.SteeringCon method*), 437
_update_plotter_db() (*mafw.processor_library.abstract_plotter.GenericPlotter method*), 390
_validate_default_type() (*mafw.processor_library.importer.FilenameElement method*), 400
_validate_filter_schema() (*in module mafw.processor.parameters*), 384
_validate_loop_type() (*mafw.tools.loop_tuning.LoopTuningMixin static method*), 484
_validate_name() (*mafw.processor.ActiveParameter method*), 353
_validate_name() (*mafw.processor.parameters.ActiveParameter method*), 382
_validate_regexp() (*mafw.processor_library.importer.FilenameElement method*), 400
_value_to_sql() (*mafw.db.trigger.Trigger method*), 187
_virtualenv_root() (*in module mafw.tools.click_extensions*), 478
_wall_clock_start (*mafw.processor.Processor attribute*), 364
_wall_clock_start (*mafw.processor.base.Processor attribute*), 377
_wire_controller_signals() (*mafw.steering_gui.main_window.MainWindow method*), 448
_write_to_path() (*mafw.steering_gui.controllers.steering_controller.S method*), 437
-D
    mafw command line option, 417
-F
    mafw-completion-install command line option, 418
--db-engine
    mafw-steering command line option, 424
--db-url
    mafw-steering command line option, 424
--debug
    mafw command line option, 417
--engine
    mafw-db-wizard command line option, 422
--ext-plugins
    mafw-steering command line option, 424
--force
    mafw-completion-install command line option, 418
--host

mafw-db-wizard command line option, 422
--ignore-unknown
mafw-db-wizard command line option, 422
--log-level
mafw command line option, 417
--no-banner
mafw command line option, 417
--no-ext-plugin
mafw-steering command line option, 424
--no-ignore-unknown
mafw-db-wizard command line option, 422
--no-open-editor
mafw-steering command line option, 424
--no-overwrite
mafw-db-wizard command line option, 422
--no-preserve-order
mafw-db-wizard command line option, 422
--no-show
mafw-steering command line option, 424
--no-snake-case
mafw-db-wizard command line option, 422
--open-editor
mafw-steering command line option, 424
--output-file
mafw-db-wizard command line option, 421
--overwrite
mafw-db-wizard command line option, 422
--password
mafw-db-wizard command line option, 422
--port
mafw-db-wizard command line option, 422
--preserve-order
mafw-db-wizard command line option, 422
--schema
mafw-db-wizard command line option, 421
--shell
mafw-completion-install command line option, 418
mafw-completion-show command line option, 419
--show
mafw-steering command line option, 424
--snake-case
mafw-db-wizard command line option, 422
--tables
mafw-db-wizard command line option, 421
--ui
mafw command line option, 417
--user
mafw-db-wizard command line option, 422
--username
mafw-db-wizard command line option, 422
--version
mafw command line option, 417
--with-views
mafw-db-wizard command line option, 422
--without-views
mafw-db-wizard command line option, 422
-e
mafw-db-wizard command line option, 422
-o
mafw-db-wizard command line option, 421
-p
mafw-db-wizard command line option, 422
-s
mafw-completion-install command line option, 418
mafw-completion-show command line option, 419
mafw-db-wizard command line option, 421
-t
mafw-db-wizard command line option, 421
-u
mafw-db-wizard command line option, 422
-v
mafw command line option, 417

A

AbbreviateGroup (*class in mafw.tools.click_extensions*), 477
Abort (*mafw.enumerators.LoopingStatus attribute*), 323
Aborted (*mafw.enumerators.ProcessorExitStatus attribute*), 323
AbortProcessorException, 341
ABOUT_FILE (*in module mafw.devtools.release.versioning*), 276
AboutDialog (*class in mafw.steering_gui.dialogs*), 442
AboutDialog (*class in mafw.steering_gui.dialogs.about_dialog*), 443
accept() (*mafw.db.db_filter.ExprParser method*), 160
accept_item() (*mafw.examples.loop_modifier.FindPrimeNumberInRan method*), 333
accept_item() (*mafw.examples.loop_modifier.FindPrimeNumberInRan method*), 334
accept_item() (*mafw.processor.base.Processor method*), 373
accept_item() (*mafw.processor.Processor method*), 359
accessor_class (*mafw.db.fields.FileChecksumField attribute*), 177
accessor_class (*mafw.db.fields.FileNameField attribute*), 178
AccumulatorProcessor (*class in mafw.examples.sum_processor*), 336
acquire_resources() (*mafw.processor.base.Processor method*), 373
acquire_resources() (*mafw.processor.Processor method*), 359
acquire_resources() (*mafw.processor.processor_list.ProcessorList method*), 386

acquire_resources() (*mafw.processor.ProcessorList method*), 366
ActionSpec (*class in mafw.steering_gui.actions*), 435
ActivableType (*class in mafw.active*), 152
Active (*class in mafw.active*), 152
ActiveParameter (*class in mafw.processor*), 351
ActiveParameter (*class in mafw.processor.parameters*), 380
ActiveType (*class in mafw.active*), 153
add_conditional() (*mafw.db.db_filter.ModelFilter method*), 166
add_conditional_from_dict() (*mafw.db.db_filter.ModelFilter method*), 166
add_group() (*mafw.steering.builder.SteeringBuilder method*), 427
add_group() (*mafw.steering_gui.controllers.steering_controller.SteeringController method*), 437
add_group_entry_to_group() (*mafw.steering_gui.controllers.steering_controller.SteeringController method*), 437
add_group_to_run_list() (*mafw.steering_gui.controllers.steering_controller.SteeringController method*), 437
add_model_filter_by_name() (*mafw.steering_gui.views.filter_tree.FilterTreeView method*), 469
add_processor() (*mafw.steering.builder.SteeringBuilder method*), 427
add_processor_to_run_list() (*mafw.steering_gui.controllers.steering_controller.SteeringController method*), 437
add_processors_to_group_with_defaults() (*mafw.steering_gui.controllers.steering_controller.SteeringController method*), 437
add_processors_with_defaults() (*mafw.steering_gui.controllers.steering_controller.SteeringController method*), 437
add_replica() (*mafw.steering.builder.SteeringBuilder method*), 427
add_replica() (*mafw.steering_gui.controllers.steering_controller.SteeringController method*), 438
add_root_logic() (*mafw.steering_gui.views.filter_tree.FilterTreeView method*), 469
add_sql() (*mafw.db.trigger.Trigger method*), 187
add_when() (*mafw.db.trigger.Trigger method*), 188
AddFieldFilterDialog (*class in mafw.steering_gui.dialogs.add_field_filter_dialog*), 443
AddModelDialog (*class in mafw.steering_gui.dialogs.add_model_dialog*), 444
AddProcessorDialog (*class in mafw.steering_gui.dialogs*), 442
AddProcessorDialog (*class in mafw.steering_gui.dialogs.add_processor_dialog*), 444
aggregation_functions (*mafw.processor_library.sns_plotter.SNSPlotter attribute*), 413
all() (*mafw.devtools.toolchain.registry.ToolRegistry method*), 298
all() (*mafw.devtools.toolchain.ToolRegistry method*), 283
always_display_progress_message (*mafw.ui.abstract_user_interface.UserInterfaceBase attribute*), 495
and_() (*in module mafw.db.trigger*), 190
append() (*mafw.processor.processor_list.ProcessorList method*), 386
append() (*mafw.processor.ProcessorList method*), 366
apply_only_to_prefix (*mafw.processor_library.db_init.TableCreator attribute*), 396
as_text_string() (*in module mafw.db.db_filter*), 168
audit (*in module mafw.devtools.cli.dependencies*), 196
AuthenticationWidget (*class in mafw.steering_gui.views.database_editor*), 461
available_db_module_count() (*mafw.steering_gui.controllers.steering_controller.SteeringController method*), 438
available_processor_count() (*mafw.steering_gui.controllers.steering_controller.SteeringController method*), 438
available_processor_names() (*mafw.steering_gui.controllers.steering_controller.SteeringController method*), 438
available_run_count() (*mafw.steering_gui.controllers.steering_controller.SteeringController method*), 438

B

BinaryNode (*in module mafw.db.db_filter*), 169
bind() (*mafw.db.db_filter.ModelFilter method*), 166
bind_all() (*mafw.db.db_filter.ProcessorFilter method*), 167
block_signals() (*in module mafw.steering_gui.utils*), 456
block_signals() (*in module mafw.steering_gui.utils.gui_helpers*), 458
bootstrap() (*mafw.devtools.toolchain.base.HostTool method*), 285
bootstrap() (*mafw.devtools.toolchain.base.ProjectTool method*), 288
bootstrap() (*mafw.devtools.toolchain.base.ToolChainTool method*), 289
bootstrap() (*mafw.devtools.toolchain.HostTool method*), 277
bootstrap() (*mafw.devtools.toolchain.ProjectTool method*), 280
bootstrap() (*mafw.devtools.toolchain.ToolChainTool method*), 281
build (*in module mafw.devtools.cli.documentation*), 197

build_connection_parameters() (*in module mafw.db.db_connection*), 154
build_connection_test_task() (*mafw.steering_gui.controllers.steering_controller.SteeringController method*), 438
build_database_connection_configuration() (*in module mafw.steering_gui.controllers.steering_controller*), 441
build_for_tag() (*in module mafw.devtools.documentation*), 221
build_for_tag() (*in module mafw.devtools.documentation.builder*), 234
build_gitlab_api_configuration() (*in module mafw.devtools.documentation*), 221
build_gitlab_api_configuration() (*in module mafw.devtools.gitlab*), 249
build_gitlab_api_configuration() (*in module mafw.devtools.gitlab.api*), 254
build_gitlab_auth_headers() (*in module mafw.devtools.documentation*), 221
build_gitlab_auth_headers() (*in module mafw.devtools.gitlab*), 249
build_gitlab_auth_headers() (*in module mafw.devtools.gitlab.api*), 254
build_pdf_for_tag() (*in module mafw.devtools.documentation*), 222
build_pdf_for_tag() (*in module mafw.devtools.documentation.builder*), 234
build_pipeline() (*mafw.steering_gui.controllers.steering_controller.SteeringController method*), 438
build_pylock_compiled_filename() (*in module mafw.devtools.gitlab*), 249
build_pylock_compiled_filename() (*in module mafw.devtools.gitlab.registry*), 258
build_pylock_reference_filename() (*in module mafw.devtools.gitlab*), 249
build_pylock_reference_filename() (*in module mafw.devtools.gitlab.registry*), 258
bump_version() (*in module mafw.devtools.release*), 263
bump_version() (*in module mafw.devtools.release.versioning*), 274

C

can_redo() (*mafw.steering_gui.controllers.steering_controller.SteeringController method*), 438
can_undo() (*mafw.steering_gui.controllers.steering_controller.SteeringController method*), 438
Category (*in module mafw.devtools.toolchain.base*), 291
category (*mafw.devtools.toolchain.base.HostTool property*), 286
category (*mafw.devtools.toolchain.base.ProjectTool property*), 289
category (*mafw.devtools.toolchain.base.ToolChainTool property*), 290
category (*mafw.devtools.toolchain.HostTool property*), 278
category (*mafw.devtools.toolchain.ProjectTool property*), 281
category (*mafw.devtools.toolchain.ToolChainTool property*), 282
CatPlot (*class in mafw.processor_library.sns_plotter*), 404
change_of_processor_status() (*mafw.ui.console_user_interface.ConsoleInterface method*), 495
change_of_processor_status() (*mafw.ui.rich_user_interface.RichInterface method*), 497
CHANGELOG_FILE (*in module mafw.devtools.release.changelog*), 270
check_ci_completion_guard() (*in module mafw.tools.click_extensions*), 479
check_main_branch() (*in module mafw.devtools.git*), 246
check_main_branch() (*in module mafw.devtools.release*), 264
check_missing_release() (*in module mafw.devtools.release*), 264
check_missing_release() (*in module mafw.devtools.release.checks*), 270
check_multiversion_structure() (*in module mafw.devtools.documentation*), 222
check_multiversion_structure() (*in module mafw.devtools.documentation.builder*), 234
class_depends_on_optional() (*in module mafw.decorators*), 191
classify_version() (*in module mafw.devtools.release*), 264
classify_version() (*in module mafw.devtools.release.versioning*), 274
clear() (*mafw.db.model_register.ModelRegister method*), 180
clear_parameters() (*mafw.steering.builder.SteeringBuilder method*), 427
clear_processor_parameters() (*mafw.steering_gui.controllers.steering_controller.SteeringCon method*), 438
closeEvent() (*mafw.steering_gui.main_window.MainWindow method*), 448
coerce_name() (*in module mafw.steering_gui.utils*), 456
coerce_name() (*in module mafw.steering_gui.utils.gui_helpers*), 458
collect_compiled_dependency_versions() (*in module mafw.devtools.dependencies*), 200
collect_compiled_dependency_versions() (*in module mafw.devtools.dependencies.compile*), 212
collect_subtree_indexes() (*mafw.steering_gui.models.filter_tree_model.FilterTreeModel method*), 451

combine_fields() (*in module mafw.tools.db_tools*), 481
combine_pk() (*in module mafw.tools.db_tools*), 481
commit_changes() (*in module mafw.devtools.git*), 246
commit_changes() (*in module mafw.devtools.release*), 264
commit_dependency_unfreeze() (*in module mafw.devtools.dependencies*), 200
commit_dependency_unfreeze() (*in module mafw.devtools.git*), 246
common_options() (*in module mafw.devtools.cli.toolchain*), 198
compare_packages() (*in module mafw.devtools.dependencies*), 200
compare_packages() (*in module mafw.devtools.dependencies.compare*), 210
compile_dependency_lockfile() (*in module mafw.devtools.dependencies*), 201
compile_dependency_lockfile() (*in module mafw.devtools.dependencies.compile*), 213
compile_python_selector() (*in module mafw.devtools.dependencies*), 201
compile_python_selector() (*in module mafw.devtools.dependencies.compile*), 213
completion_script_path() (*in module mafw.tools.click_extensions*), 479
completion_shell_from_env() (*in module mafw.tools.click_extensions*), 479
completion_source_script() (*in module mafw.tools.click_extensions*), 479
compute_doc_target_version() (*in module mafw.devtools.release.versioning*), 274
compute_upper_bound() (*in module mafw.devtools.dependencies*), 201
compute_upper_bound() (*in module mafw.devtools.dependencies.freeze*), 216
Condition (*class in mafw.steering.models*), 430
condition_key_for_index() (*mafw.steering_gui.models.filter_tree_model.FilterTreeModel method*), 451
condition_key_for_item() (*mafw.steering_gui.models.filter_tree_model.FilterTreeModel method*), 451
condition_key_for_parent() (*mafw.steering_gui.models.filter_tree_model.FilterTreeModel method*), 451
conditional_name (*mafw.db.db_filter.ModelFilter attribute*), 167
ConditionalFilterCondition (*class in mafw.db.db_filter*), 158
ConditionalFilterConfig (*class in mafw.steering.models*), 430
ConditionalFilterEditor (*class in mafw.steering_gui.views.filter_editors*), 465
ConditionalNode (*class in mafw.db.db_filter*), 159
ConditionDefinition (*class in mafw.steering_gui.views.filter_editors*), 464
ConditionEditor (*class in mafw.steering_gui.views.filter_editors*), 465
ConditionKey (*class in mafw.steering_gui.models.filter_tree_model*), 450
ConditionNode (*class in mafw.db.db_filter*), 157
ConditionsTableModel (*class in mafw.steering_gui.views.filter_editors*), 465
connect() (*mafw.steering_gui.controllers.steering_controller.DirtyState method*), 437
connect() (*mafw.steering_gui.threading.Worker method*), 454
connect() (*mafw.steering_gui.threading.workers.Worker method*), 455
CONSOLE (*in module mafw.tools.shell_tools*), 489
ConsoleInterface (*class in mafw.ui.console_user_interface*), 495
consume_value() (*mafw.devtools.release.DeprecatedOption method*), 263
consume_value() (*mafw.tools.click_extensions.DeprecatedOption method*), 478
consumer_finish() (*mafw.processor.base.Processor method*), 373
consumer_finish() (*mafw.processor.Processor method*), 359
consumer_process() (*mafw.processor.base.Processor method*), 373
consumer_process() (*mafw.processor.Processor method*), 359
consumer_start() (*mafw.processor.base.Processor method*), 373
consumer_start() (*mafw.processor.Processor method*), 359
CONTEXT_SETTINGS (*in module mafw.devtools.cli.dependencies*), 196
CONTEXT_SETTINGS (*in module mafw.devtools.cli.documentation*), 197
CONTEXT_SETTINGS (*in module mafw.devtools.cli.release*), 198
Continue (*mafw.enumerators.LoopingStatus attribute*), 323
copy_patch_files() (*in module mafw.devtools.documentation*), 222
copy_patch_files() (*in module mafw.devtools.documentation.builder*), 235
CountStandardTables (*class in mafw.examples.db_processors*), 324
create (*in module mafw.devtools.cli.release*), 198
create() (*mafw.db.trigger.Trigger method*), 188
create_action() (*in module*

mafw.steering_gui.actions), 435
create_docs_zip_for_tag() (*in module mafw.devtools.documentation*), 222
create_docs_zip_for_tag() (*in module mafw.devtools.documentation.builder*), 235
create_release_note() (*in module mafw.devtools.release*), 264
create_release_note() (*in module mafw.devtools.release.notes*), 271
create_standard_tables (*mafw.processor.base.Processor attribute*), 377
create_standard_tables (*mafw.processor.Processor attribute*), 364
create_standard_tables (*mafw.processor.processor_list.ProcessorList attribute*), 386
create_standard_tables (*mafw.processor.ProcessorList attribute*), 366
create_table() (*mafw.db.db_model.MAFwBaseModel class method*), 171
create_tag() (*in module mafw.devtools.git*), 246
create_tag() (*in module mafw.devtools.release*), 265
create_task() (*mafw.ui.abstract_user_interface.UserInterfaceBase method*), 494
create_task() (*mafw.ui.console_user_interface.ConsoleInterface method*), 496
create_task() (*mafw.ui.rich_user_interface.RichInterface method*), 497
create_trigger_sql() (*mafw.db.trigger.MySQLDialect method*), 185
create_trigger_sql() (*mafw.db.trigger.PostgreSQLDialect method*), 185
create_trigger_sql() (*mafw.db.trigger.SQLiteDialect method*), 186
create_trigger_sql() (*mafw.db.trigger.TriggerDialect method*), 189
current_file() (*mafw.steering_gui.controllers.steering_controller.SteeringController method*), 438
current_name() (*mafw.steering_gui.views.processor_parameter_editor.ProcessorParameterEditor method*), 473
current_name() (*mafw.steering_gui.views.ProcessorParameterEditor method*), 460
current_table_values() (*mafw.steering_gui.views.database_editor.ParameterWidget method*), 463
current_values() (*mafw.steering_gui.views.globals_editor.GlobalsEditor method*), 470
current_values() (*mafw.steering_gui.views.GlobalsEditor method*), 459
custom_formatwarning() (*in module mafw.scripts.mafw_exe*), 425
customize_plot() (*mafw.processor_library.abstract_plotter.GenericPl method*), 391
CyclicGroupIssue, 341

D

data_frame (*mafw.processor_library.sns_plotter.SNSFigurePlotter attribute*), 410
data_frame (*mafw.processor_library.sns_plotter.SNSPlotter attribute*), 413
DATABASE
    mafw-db-wizard command line option, 422
database (*mafw.db.trigger.Trigger property*), 189
database (*mafw.processor.base.Processor property*), 377
database (*mafw.processor.Processor property*), 364
database (*mafw.processor.processor_list.ProcessorList property*), 386
database (*mafw.processor.ProcessorList property*), 366
database (*mafw.processor_library.sns_plotter.SQLPdDataRetriever attribute*), 413
database_proxy (*in module mafw.db.db_model*), 175
database_required() (*in module mafw.decorators*), 192
DatabaseEditor (*class in mafw.steering_gui.views*), 458
DatabaseEditor (*class in mafw.steering_gui.views.database_editor*), 462
DataRetriever (*class in mafw.processor_library.abstract_plotter*), 388
db_model_modules (*mafw.plugin_manager.LoadedPlugins attribute*), 348
db_scheme (*in module mafw.db.db_configurations*), 154
db_value() (*mafw.db.fields.FileChecksumField method*), 178
db_value() (*mafw.db.fields.FileNameField method*), 179
db_value() (*mafw.db.fields.FileNameListField method*), 179
DBConfiguration (*class in mafw.steering.models*), 430
deep_update() (*in module mafw.tools.generics*), 483
DEFAULT_BACKEND (*in module mafw.steering_gui.views.database_editor*), 465
default_conf (*in module mafw.db.db_configurations*), 154
DEFAULT_FREEZE_EXTRAS (*in module mafw.devtools.dependencies.compile*), 215
DEFAULT_MIN_TAG_REGEX (*in module mafw.devtools.documentation.builder*), 238
DEFAULT_PRAGMAS (*in module mafw.steering_gui.views.database_editor*),

463
DEFAULT_SERVER_PORTS (*in module mafw.steering_gui.views.database_editor*), 463
DEFAULT_SQLITE_PRAGMAS (*in module mafw.db.db_configurations*), 154
DEFAULT_URL_PLACEHOLDERS (*in module mafw.steering_gui.views.database_editor*), 464
DEFAULT_URLS (*in module mafw.steering_gui.views.database_editor*), 464
delete_generic_package_file() (*in module mafw.devtools.gitlab*), 249
delete_generic_package_file() (*in module mafw.devtools.gitlab.registry*), 258
delete_generic_package_version() (*in module mafw.devtools.gitlab*), 250
delete_generic_package_version() (*in module mafw.devtools.gitlab.registry*), 259
delete_parameter() (*mafw.processor.base.Processor method*), 373
delete_parameter() (*mafw.processor.Processor method*), 360
depends_on_optional() (*in module mafw.decorators*), 192
DeprecatedOption (*class in mafw.devtools.release*), 262
DeprecatedOption (*class in mafw.tools.click_extensions*), 477
description (*mafw.processor.base.Processor attribute*), 378
description (*mafw.processor.Processor attribute*), 364
detect_current_version() (*mafw.devtools.toolchain.base.HostTool method*), 285
detect_current_version() (*mafw.devtools.toolchain.base.ProjectTool method*), 288
detect_current_version() (*mafw.devtools.toolchain.base.ToolChainTool method*), 290
detect_current_version() (*mafw.devtools.toolchain.HostTool method*), 278
detect_current_version() (*mafw.devtools.toolchain.ProjectTool method*), 280
detect_current_version() (*mafw.devtools.toolchain.ToolChainTool method*), 282
detect_latest_version() (*mafw.devtools.toolchain.base.HostTool method*), 286
detect_latest_version() (*mafw.devtools.toolchain.base.ProjectTool method*), 288
detect_latest_version() (*mafw.devtools.toolchain.base.ToolChainTool method*), 290
detect_latest_version() (*mafw.devtools.toolchain.HostTool method*), 278
detect_latest_version() (*mafw.devtools.toolchain.ProjectTool method*), 280
detect_latest_version() (*mafw.devtools.toolchain.ToolChainTool method*), 282
DevtoolsError, 195
dirty (*mafw.steering_gui.controllers.steering_controller.SteeringContro property*), 441
DirtyStateSignal (*class in mafw.steering_gui.controllers.steering_controller*), 436
disable() (*mafw.db.std_tables.TriggerDisabler method*), 184
disable_db_configuration() (*mafw.steering.builder.SteeringBuilder method*), 427
disable_db_configuration() (*mafw.steering_gui.controllers.steering_controller.SteeringCon method*), 438
display_exception() (*in module mafw.scripts.mafw_exe*), 425
display_progress_message() (*mafw.ui.abstract_user_interface.UserInterfaceBase method*), 494
display_progress_message() (*mafw.ui.console_user_interface.ConsoleInterface method*), 496
display_progress_message() (*mafw.ui.rich_user_interface.RichInterface method*), 498
DisPlot (*class in mafw.processor_library.sns_plotter*), 406
distribute_resources() (*mafw.processor.processor_list.ProcessorList method*), 386
distribute_resources() (*mafw.processor.ProcessorList method*), 366
DOC_TARGET_VERSION_PATTERN (*in module mafw.devtools.release.versioning*), 276
DOCS_SUBPATH (*in module mafw.devtools.documentation.builder*), 239
DocTargetVersionPrompt (*class in mafw.devtools.release*), 263
DocTargetVersionPrompt (*class in mafw.devtools.release.versioning*), 273
document (*mafw.steering.builder.SteeringBuilder property*), 429
DoesNotExist (*mafw.db.db_model.MAFwBaseModel*

attribute), 171
DoesNotExist (*mafw.db.std_tables.OrphanFile attribute*), 183
DoesNotExist (*mafw.db.std_tables.PlotterOutput attribute*), 183
DoesNotExist (*mafw.db.std_tables.StandardTable attribute*), 183
DoesNotExist (*mafw.db.std_tables.TriggerStatus attribute*), 184
DoesNotExist (*mafw.examples.db_processors.File attribute*), 325
DoesNotExist (*mafw.examples.importer_example.InputFile attribute*), 328
download_docs_zip_from_gitlab_generic_registry() (*in module mafw.devtools.documentation*), 223
download_docs_zip_from_gitlab_generic_registry() (*in module mafw.devtools.gitlab.docs_registry*), 255
download_generic_file() (*in module mafw.devtools.gitlab*), 250
download_generic_file() (*in module mafw.devtools.gitlab.registry*), 259
drop() (*mafw.db.trigger.Trigger method*), 188
drop_trigger_sql() (*mafw.db.trigger.MySQLDialect method*), 185
drop_trigger_sql() (*mafw.db.trigger.PostgreSQLDialect method*), 186
drop_trigger_sql() (*mafw.db.trigger.SQLiteDialect method*), 186
drop_trigger_sql() (*mafw.db.trigger.TriggerDialect method*), 189
dump_models() (*in module mafw.db.db_wizard*), 176
dump_parameter_configuration() (*mafw.processor.base.Processor method*), 373
dump_parameter_configuration() (*mafw.processor.Processor method*), 360
dump_processor_parameters_to_toml() (*in module mafw.tools.toml_tools*), 490
DuplicateReplicaIssue, 341
duration (*mafw.models.loop_payloads.LoopResult attribute*), 346
duration (*mafw.models.LoopResult attribute*), 344
duration (*mafw.timer.Timer property*), 475

E

elements (*mafw.processor_library.importer.FilenameParser property*), 402
emit() (*mafw.steering_gui.controllers.steering_controller.DirtyStateSignal method*), 437
enable() (*mafw.db.std_tables.TriggerDisabler method*), 184
enable_db_configuration() (*mafw.steering.builder.SteeringBuilder method*), 427
enable_db_configuration() (*mafw.steering_gui.controllers.steering_controller.SteeringCon method*), 438
enabled (*mafw.steering_gui.actions.ActionSpec attribute*), 435
ensure_clean_git() (*in module mafw.devtools.git*), 247
ensure_clean_git() (*in module mafw.devtools.release*), 265
ensure_devtools_available() (*in module mafw.devtools*), 195
ensure_mafw_project_root() (*in module mafw.devtools.dependencies*), 202
ensure_mafw_project_root() (*in module mafw.devtools.dependencies.compile*), 213
ensure_parameter_registration() (*in module mafw.processor*), 367
ensure_parameter_registration() (*in module mafw.processor.parameters*), 384
ensure_sphinx_build_available() (*in module mafw.devtools.documentation*), 224
ensure_sphinx_build_available() (*in module mafw.devtools.documentation.builder*), 235
ensure_valid() (*mafw.steering_gui.controllers.steering_controller.Stee method*), 438
ensure_versions_json_exists() (*in module mafw.devtools.documentation*), 224
ensure_versions_json_exists() (*in module mafw.devtools.documentation.versions*), 244
enter_interactive_mode() (*mafw.ui.abstract_user_interface.UserInterfaceBase method*), 494
enter_interactive_mode() (*mafw.ui.rich_user_interface.RichInterface method*), 498
ENV_ESCAPE_SENTINEL (*in module mafw.tools.toml_tools*), 493
env_name (*mafw.devtools.toolchain.base.ProjectTool property*), 289
env_name (*mafw.devtools.toolchain.ProjectTool property*), 281
env_name (*mafw.devtools.toolchain.tools.git_cliff.GitCliffTool property*), 308
env_name (*mafw.devtools.toolchain.tools.GitCliffTool property*), 299
env_name (*mafw.devtools.toolchain.tools.mypy.MypyTool property*), 310
env_name (*mafw.devtools.toolchain.tools.MypyTool property*), 301
env_name (*mafw.devtools.toolchain.tools.pip_audit.PipAuditTool property*), 310
env_name (*mafw.devtools.toolchain.tools.PipAuditTool property*), 301
env_name (*mafw.devtools.toolchain.tools.precommit.PreCommitTool property*), 312

env_name (*mafw.devtools.toolchain.tools.PreCommitTool property*), 302
env_name (*mafw.devtools.toolchain.tools.pytest.PytestTool property*), 313
env_name (*mafw.devtools.toolchain.tools.PytestTool property*), 303
env_name (*mafw.devtools.toolchain.tools.pyupgrade.PyUpgradeTool property*), 316
env_name (*mafw.devtools.toolchain.tools.ruff.RuffTool property*), 318
env_name (*mafw.devtools.toolchain.tools.RuffTool property*), 305
env_name (*mafw.devtools.toolchain.tools.sphinx.SphinxTool property*), 319
env_name (*mafw.devtools.toolchain.tools.SphinxTool property*), 306
ENV_PATTERN (*in module mafw.tools.toml_tools*), 493
Error (*mafw.scripts.mafw_exe.ReturnValue attribute*), 425
execute() (*mafw.processor.base.Processor method*), 374
execute() (*mafw.processor.Processor method*), 360
execute() (*mafw.processor.processor_list.ProcessorList method*), 386
execute() (*mafw.processor.ProcessorList method*), 366
execution_workflow() (*in module mafw.decorators*), 193
existing_table_names (*mafw.processor_library.db_init.TableCreator attribute*), 396
exit_status (*mafw.runner.MAFwApplication attribute*), 416
expect() (*mafw.db.db_filter.ExprParser method*), 160
export_to_path() (*mafw.steering_gui.controllers.steering_controller.SteeringController method*), 438
ExprNode (*in module mafw.db.db_filter*), 169
ExprParser (*class in mafw.db.db_filter*), 159
extend() (*mafw.processor.processor_list.ProcessorList method*), 386
extend() (*mafw.processor.ProcessorList method*), 366
EXTENSION_FORMAT_MAP (*in module mafw.devtools.dependencies.compare*), 212
extra_globals (*mafw.steering.builder.SteeringBuilder property*), 429
extract_change_sections_from_changelog() (*in module mafw.devtools.release*), 265
extract_change_sections_from_changelog() (*in module mafw.devtools.release.changelog*), 269
extract_change_sections_from_changelog_block() (*in module mafw.devtools.release*), 265
extract_change_sections_from_changelog_block() (*in module mafw.devtools.release.changelog*), 269
extract_docs_zip_to_repo_root() (*in module mafw.devtools.documentation*), 224
extract_docs_zip_to_repo_root() (*in module mafw.devtools.documentation.builder*), 235
extract_protocol() (*in module mafw.tools.regexp*), 486
extract_version_changelog_block() (*in module mafw.devtools.release*), 265
extract_version_changelog_block() (*in module mafw.devtools.release.changelog*), 270

F

F (*class in mafw.decorators*), 191
facet_grid (*mafw.processor_library.sns_plotter.SNSFigurePlotter attribute*), 410
facet_grid (*mafw.processor_library.sns_plotter.SNSPlotter attribute*), 413
Failed (*mafw.enumerators.ProcessorExitStatus attribute*), 323
fetch_latest_version() (*in module mafw.devtools.toolchain.pypi*), 295
FieldFilterConfig (*class in mafw.steering.models*), 430
FieldFilterEditor (*class in mafw.steering_gui.views.filter_editors*), 465
FieldInfo (*class in mafw.steering.models*), 431
FieldNameWidget (*class in mafw.steering_gui.views.filter_editors*), 465
FigurePlotter (*class in mafw.processor_library.abstract_plotter*), 388
File (*class in mafw.examples.db_processors*), 325
file_browse_button (*mafw.steering_gui.views.database_editor.AuthenticationWidge property*), 462
file_checksum() (*in module mafw.tools.file_tools*), 482
file_edit (*mafw.steering_gui.views.database_editor.AuthenticationWid property*), 462
file_removal_triggers() (*mafw.db.db_model.MAFwBaseModel class method*), 171
FileChecksumField (*class in mafw.db.fields*), 177
FileChecksumFieldAccessor (*class in mafw.db.fields*), 178
FileDoesNotExist, 324
FilenameElement (*class in mafw.processor_library.importer*), 399
FileNameField (*class in mafw.db.fields*), 178
FileNameFieldAccessor (*class in mafw.db.fields*), 179
FileNameListField (*class in mafw.db.fields*), 179
FilenameParser (*class in mafw.processor_library.importer*), 401
fill_model_info() (*mafw.steering.models.ConditionalFilterConfig method*), 430
fill_model_info() (*mafw.steering.models.ModelFilterConfig method*), 431

FillFileTableProcessor (*class in mafw.examples.db_processors*), 325
filter() (*mafw.db.db_filter.ModelFilter method*), 166
filter_all() (*mafw.db.db_filter.ProcessorFilter method*), 168
filter_by_category() (*mafw.devtools.toolchain.registry.ToolRegistry method*), 298
filter_by_category() (*mafw.devtools.toolchain.ToolRegistry method*), 283
filter_latest_micro() (*in module mafw.devtools.documentation*), 224
filter_latest_micro() (*in module mafw.devtools.documentation.builder*), 236
filter_logic (*mafw.steering.models.ProcessorConfig property*), 433
filter_register (*mafw.processor.base.Processor attribute*), 378
filter_register (*mafw.processor.Processor attribute*), 364
filter_schema() (*mafw.lazy_import.ProcessorClassProtocol method*), 339
filter_schema() (*mafw.processor.base.Processor class method*), 369
filter_schema() (*mafw.processor.Processor class method*), 355
filter_stable_tags() (*in module mafw.devtools.documentation*), 225
filter_stable_tags() (*in module mafw.devtools.documentation.builder*), 236
filter_versions_in_range() (*in module mafw.devtools.documentation*), 225
filter_versions_in_range() (*in module mafw.devtools.documentation.builder*), 236
FilterConfig (*class in mafw.steering.models*), 431
FilterKind (*class in mafw.steering.models*), 431
FilterManager (*class in mafw.steering_gui.views.filter_manager*), 469
FilterNode (*class in mafw.db.db_filter*), 161
FilterNodeType (*class in mafw.steering_gui.models.filter_tree_model*), 450
FilterSchema (*class in mafw.models*), 344
FilterSchema (*class in mafw.models.filter_schema*), 345
FilterTreeItem (*class in mafw.steering_gui.models.filter_tree_model*), 450
FilterTreeModel (*class in mafw.steering_gui.models.filter_tree_model*), 451
FilterTreeView (*class in mafw.steering_gui.views.filter_tree*), 469
find_condition_index() (*mafw.steering_gui.models.filter_tree_model.FilterTreeModel method*), 451
find_index_for_data() (*mafw.steering_gui.models.filter_tree_model.FilterTreeModel method*), 451
find_logic_index() (*mafw.steering_gui.models.filter_tree_model.FilterTreeModel method*), 451
find_model_index() (*mafw.steering_gui.models.filter_tree_model.FilterTreeModel method*), 451
find_repo_root() (*in module mafw.devtools.documentation*), 225
find_repo_root() (*in module mafw.devtools.documentation.builder*), 236
FindNPrimeNumber (*class in mafw.examples.loop_modifier*), 329
FindPrimeNumberInRange (*class in mafw.examples.loop_modifier*), 330
FindPrimeNumberInRangeParallel (*class in mafw.examples.loop_modifier*), 332
FindPrimeNumberInRangeParallelWithQueue (*class in mafw.examples.loop_modifier*), 332
FindPrimeNumberInRangeTunable (*class in mafw.examples.loop_modifier*), 333
Finish (*mafw.enumerators.ProcessorStatus attribute*), 323
finish() (*mafw.examples.db_processors.FillFileTableProcessor method*), 326
finish() (*mafw.examples.importer_example.ImporterExample method*), 328
finish() (*mafw.examples.loop_modifier.FindNPrimeNumber method*), 329
finish() (*mafw.examples.loop_modifier.FindPrimeNumberInRange method*), 331
finish() (*mafw.examples.loop_modifier.FindPrimeNumberInRangePara method*), 332
finish() (*mafw.examples.loop_modifier.FindPrimeNumberInRangePara method*), 333
finish() (*mafw.examples.loop_modifier.FindPrimeNumberInRangeTuna method*), 334
finish() (*mafw.processor.base.Processor method*), 374
finish() (*mafw.processor.Processor method*), 360
finish() (*mafw.processor_library.abstract_plotter.GenericPlotter method*), 391
finish() (*mafw.processor_library.db_init.TriggerRefresher method*), 398
FLOAT_DECIMALS (*in module mafw.steering_gui.utils.constants*), 456
FLOAT_RANGE (*in module mafw.steering_gui.utils.constants*), 456
for_loop() (*in module mafw.decorators*), 193
force_recreate (*mafw.processor_library.db_init.TableCreator attribute*), 396
force_replot (*mafw.processor_library.abstract_plotter.GenericPlotter*

attribute), 392
ForLoop (*mafw.enumerators.LoopType attribute*), 322
format_duration() (*mafw.timer.Timer method*), 475
format_progress_message() (*mafw.examples.db_processors.FillFileTableProcessor method*), 326
format_progress_message() (*mafw.examples.importer_example.ImporterExample method*), 328
format_progress_message() (*mafw.examples.loop_modifier.FindNPrimeNumbers method*), 329
format_progress_message() (*mafw.examples.loop_modifier.FindPrimeNumberInRange method*), 331
format_progress_message() (*mafw.examples.loop_modifier.FindPrimeNumberInRangeDynamic method*), 332
format_progress_message() (*mafw.examples.loop_modifier.FindPrimeNumberInRangeParallel method*), 333
format_progress_message() (*mafw.examples.loop_modifier.FindPrimeNumberInRangeWithQueue method*), 334
format_progress_message() (*mafw.processor.base.Processor method*), 374
format_progress_message() (*mafw.processor.Processor method*), 360
format_progress_message() (*mafw.processor_library.abstract_plotter.GenericPlotter method*), 391
format_progress_message() (*mafw.processor_library.db_init.SQLScriptRunner method*), 394
format_progress_message() (*mafw.processor_library.db_init.TriggerRefresher method*), 398
format_progress_message() (*mafw.processor_library.importer.Importer method*), 404
format_requirement() (*in module mafw.devtools.dependencies.freeze*), 216
format_size() (*in module mafw.devtools.documentation*), 225
format_size() (*in module mafw.devtools.documentation.versions*), 244
format_started_at_utc() (*in module mafw.devtools.documentation*), 225
format_started_at_utc() (*in module mafw.devtools.documentation.server*), 241
FreeNameWidget (*class in mafw.steering_gui.views.filter_editors*), 465
freeze (*in module mafw.devtools.cli.dependencies*), 197
freeze_dependencies() (*in module mafw.devtools.dependencies*), 202
freeze_dependencies() (*in module mafw.devtools.dependencies.freeze*), 216
freeze_pyproject_toml() (*in module mafw.devtools.dependencies*), 202
freeze_pyproject_toml() (*in module mafw.devtools.dependencies.freeze*), 217
freeze_requirement() (*in module mafw.devtools.dependencies*), 203
freeze_requirement() (*in module mafw.devtools.dependencies.freeze*), 217
from_conf() (*mafw.db.db_filter.ModelFilter class method*), 165
from_dict() (*mafw.db.db_model.MAFwBaseModel class method*), 172
from_dict() (*mafw.processor_library.importer.FilenameElement class method*), 400
from_file() (*mafw.steering_gui.controllers.steering_controller.SteeringController method*), 437
from_toml() (*mafw.steering.builder.SteeringBuilder class method*), 427
from_toml_text() (*mafw.steering.builder.SteeringBuilder class method*), 427
FromDatasetDataRetriever (*class in mafw.processor_library.sns_plotter*), 407
full_name (*mafw.steering.models.ProcessorRef property*), 433

G

GaussAdder (*class in mafw.examples.sum_processor*), 336
generate_changelog() (*in module mafw.devtools.release*), 266
generate_changelog() (*in module mafw.devtools.release.changelog*), 270
generate_pdf_index_page() (*in module mafw.devtools.documentation*), 226
generate_pdf_index_page() (*in module mafw.devtools.documentation.builder*), 237
generate_python_versions_rst() (*in module mafw.devtools.documentation*), 226
generate_python_versions_rst() (*in module mafw.devtools.documentation.requirements*), 240
generate_requirements_rst() (*in module mafw.devtools.documentation*), 226
generate_requirements_rst() (*in module mafw.devtools.documentation.requirements*), 240
generate_steering_file() (*in module mafw.tools.toml_tools*), 491
GenericPlotter (*class in mafw.processor_library.abstract_plotter*), 388
get() (*mafw.devtools.toolchain.registry.ToolRegistry method*), 299

get() (*mafw.devtools.toolchain.ToolRegistry method*), 283
get_analysis_description() (*mafw.steering_gui.controllers.steering_controller.SteeringController method*), 438
get_analysis_name() (*mafw.steering_gui.controllers.steering_controller.SteeringController method*), 438
get_command() (*mafw.tools.click_extensions.AbbreviateGroup method*), 477
get_contributors() (*in module mafw.devtools.release*), 266
get_contributors() (*in module mafw.devtools.release.notes*), 271
get_create_standard_tables() (*mafw.steering_gui.controllers.steering_controller.SteeringController method*), 438
get_current_branch() (*in module mafw.devtools.documentation*), 226
get_current_branch() (*in module mafw.devtools.git*), 247
get_data() (*mafw.steering_gui.views.database_editor.AdvancedFilterWidget method*), 461
get_data() (*mafw.steering_gui.views.database_editor.DatabaseEditor method*), 462
get_data() (*mafw.steering_gui.views.database_editor.ParametersWidget method*), 463
get_data() (*mafw.steering_gui.views.DatabaseEditor method*), 459
get_data_frame() (*mafw.processor_library.abstract_plotter.DataRetriever method*), 388
get_data_frame() (*mafw.processor_library.abstract_plotter.GlobalPlotter method*), 391
get_data_frame() (*mafw.processor_library.sns_plotter.FacetGridDataRetriever method*), 407
get_data_frame() (*mafw.processor_library.sns_plotter.HDFPdDataRetriever method*), 407
get_data_frame() (*mafw.processor_library.sns_plotter.PdDataRetriever method*), 408
get_data_frame() (*mafw.processor_library.sns_plotter.SNSPlotter method*), 412
get_data_frame() (*mafw.processor_library.sns_plotter.SQLPdDataRetriever method*), 413
get_db() (*in module mafw.db.db_model*), 174
get_db_attributes() (*mafw.steering_gui.controllers.steering_controller.SteeringController method*), 438
get_db_authentication() (*mafw.steering_gui.controllers.steering_controller.SteeringController method*), 439
get_db_parameters() (*mafw.steering_gui.controllers.steering_controller.SteeringController method*), 439
get_db_pragmas() (*mafw.steering_gui.controllers.steering_controller.SteeringController method*), 439
get_db_url() (*mafw.steering_gui.controllers.steering_controller.SteeringController method*), 439
get_default_registry() (*in module mafw.devtools.toolchain.default_registry*), 291
get_dependency_specifier() (*mafw.devtools.toolchain.pyproject_modifier.PyprojectModifier method*), 296
get_dialect() (*mafw.processor_library.db_init.TriggerRefresher method*), 298
get_directory_size() (*in module mafw.devtools.documentation*), 226
get_directory_size() (*in module mafw.devtools.documentation.versions*), 244
get_element() (*mafw.processor_library.importer.FilenameParser method*), 402
get_element_value() (*mafw.processor_library.importer.FilenameParser method*), 402
get_expected_lower_bounds() (*in module mafw.devtools.dependencies*), 203
get_expected_lower_bounds() (*in module mafw.devtools.dependencies.verify*), 220
get_field_type() (*mafw.db.db_model.MAFwBaseModel class method*), 172
get_filter() (*mafw.processor.base.Processor method*), 374
get_filter() (*mafw.processor.Processor method*), 360
get_git_tags() (*in module mafw.devtools.documentation*), 227
get_git_tags() (*in module mafw.devtools.git*), 247
get_global_plot() (*mafw.steering_gui.controllers.steering_controller.SteeringController method*), 439
get_group_data() (*mafw.steering.builder.SteeringBuilder method*), 427
get_group_attributes() (*mafw.steering_gui.controllers.steering_controller.SteeringController method*), 439
get_group_snapshot() (*mafw.steering_gui.controllers.steering_controller.SteeringController method*), 439
get_items() (*mafw.examples.db_processors.FillFileTableProcessor method*), 326
get_items() (*mafw.examples.importer_example.ImporterExample method*), 328
get_items() (*mafw.examples.loop_modifier.FindPrimeNumberInRange method*), 331
get_items() (*mafw.examples.loop_modifier.FindPrimeNumberInRange method*), 332
get_items() (*mafw.examples.loop_modifier.FindPrimeNumberInRange method*), 333
get_items() (*mafw.examples.loop_modifier.FindPrimeNumberInRange method*), 334
get_items() (*mafw.examples.loop_modifier.ModifyLoopProcessor method*), 335
get_items() (*mafw.examples.sum_processor.AccumulatorProcessor method*), 336
get_items() (*mafw.processor.base.Processor*

method), 374
get_items() (*mafw.processor.Processor method*), 361
get_items() (*mafw.processor_library.db_init.SQLScriptExecutor method*), 394
get_items() (*mafw.processor_library.db_init.TriggerRefresher method*), 398
get_last_stable_tag() (*in module mafw.devtools.git*), 247
get_last_stable_tag() (*in module mafw.devtools.release*), 266
get_model() (*mafw.db.model_register.ModelRegister method*), 180
get_model_name() (*mafw.steering_gui.views.filter_editors.ProcessedTableModel method*), 467
get_model_names() (*mafw.db.model_register.ModelRegister method*), 181
get_name() (*mafw.steering_gui.views.filter_editors.FieldNameWidget method*), 465
get_name() (*mafw.steering_gui.views.filter_editors.FreeNameWidget method*), 466
get_new_only() (*mafw.steering_gui.controllers.steering_controller.SteeringController method*), 439
get_parameter() (*mafw.models.processor_schema.ProcessorSchema method*), 347
get_parameter() (*mafw.models.ProcessorSchema method*), 345
get_parameter() (*mafw.processor.base.Processor method*), 375
get_parameter() (*mafw.processor.Processor method*), 361
get_parameter_names() (*mafw.models.processor_schema.ProcessorSchema method*), 347
get_parameter_names() (*mafw.models.ProcessorSchema method*), 345
get_parameters() (*mafw.processor.base.Processor method*), 375
get_parameters() (*mafw.processor.Processor method*), 361
get_pk() (*in module mafw.tools.db_tools*), 481
get_plugin_manager() (*in module mafw.plugin_manager*), 351
get_primary_key_fields() (*in module mafw.db.db_model*), 175
get_processor_config() (*mafw.steering.builder.SteeringBuilder method*), 427
get_processor_snapshot() (*mafw.steering_gui.controllers.steering_controller.SteeringController method*), 439
get_processors_to_run() (*mafw.steering_gui.controllers.steering_controller.SteeringController method*), 439
get_release_note_base_ref() (*in module mafw.devtools.release*), 266
get_release_note_base_ref() (*in module mafw.devtools.release.notes*), 272
get_release_statistics() (*in module mafw.devtools.release*), 266
get_release_statistics() (*in module mafw.devtools.release.notes*), 272
get_server_status() (*in module mafw.devtools.documentation.server*), 241
get_standard_tables() (*mafw.db.model_register.ModelRegister method*), 181
get_table_names() (*mafw.db.model_register.ModelRegister method*), 181
get_traceback_exception_handler() (*in module mafw.steering_gui.utils.exception_handling*), 457
get_type() (*mafw.steering_gui.views.filter_editors.FieldNameWidget method*), 465
get_type() (*mafw.steering_gui.views.filter_editors.FreeNameWidget method*), 466
get_ui_interface() (*mafw.steering_gui.controllers.steering_controller.SteeringController method*), 439
get_url() (*mafw.steering_gui.views.database_editor.TransportWidget method*), 463
get_user_interface() (*mafw.runner.MAFwApplication method*), 415
get_value() (*mafw.steering_gui.views.filter_editors.ListEditor method*), 466
get_value() (*mafw.steering_gui.views.filter_editors.RangeEditor method*), 467
get_value() (*mafw.steering_gui.views.filter_editors.SingleValueEditor method*), 467
get_value() (*mafw.steering_gui.views.filter_editors.ValueEditorBase method*), 468
git_rev_of() (*in module mafw.devtools.documentation*), 227
git_rev_of() (*in module mafw.devtools.git*), 247
GitCliffTool (*class in mafw.devtools.toolchain.tools*), 299
GitCliffTool (*class in mafw.devtools.toolchain.tools.git_cliff*), 307
GitlabAPIConfiguration (*class in mafw.devtools.gitlab*), 248
GitlabAPIConfiguration (*class in mafw.devtools.gitlab.api*), 253
global_mafw_plugin_manager (*in module mafw.plugin_manager*), 351
GlobalSettingsEditor (*class in mafw.steering_gui.views*), 459
GlobalsEditor (*class in mafw.steering_gui.views.globals_editor*), 470
GlobalSettings (*class in mafw.steering.models*), 431
group_and_aggregate_data_frame() (*in module mafw.tools.pandas_tools*), 485

group_and_aggregate_data_frame() (*mafw.processor_library.sns_plotter.SNSPlotter method*), 412
group_class (*mafw.scripts.mafw_exe.MAFwGroup attribute*), 424
group_class (*mafw.tools.click_extensions.AbbreviateGroup attribute*), 477
GroupConfig (*class in mafw.steering.models*), 431
GroupEditor (*class in mafw.steering_gui.views*), 459
GroupEditor (*class in mafw.steering_gui.views.group_editor*), 470
grouping_columns (*mafw.processor_library.sns_plotter.SNSPlotter attribute*), 413

H

handle_exception() (*mafw.steering_gui.utils.exception_handling.ThreadExceptionHandler method*), 457
has_changes (*mafw.devtools.dependencies.compare.VersionComparisonResult property*), 210
has_changes (*mafw.devtools.dependencies.VersionComparisonResult property*), 200
has_frozen_upper_bound() (*in module mafw.devtools.dependencies.freeze*), 218
has_required_fields() (*mafw.steering_gui.views.database_editor.AuthenticationWidget method*), 462
HatchTool (*class in mafw.devtools.toolchain.tools*), 300
HatchTool (*class in mafw.devtools.toolchain.tools.hatch*), 308
HDFPdDataRetriever (*class in mafw.processor_library.sns_plotter*), 407
highest_lower_bound() (*in module mafw.devtools.dependencies.freeze*), 218
HostTool (*class in mafw.devtools.toolchain*), 277
HostTool (*class in mafw.devtools.toolchain.base*), 285
http_probe() (*in module mafw.devtools.documentation*), 227
http_probe() (*in module mafw.devtools.documentation.server*), 242

I

i_item (*mafw.models.loop_payloads.LoopItem attribute*), 346
i_item (*mafw.models.loop_payloads.LoopResult attribute*), 346
i_item (*mafw.models.LoopItem attribute*), 344
i_item (*mafw.models.LoopResult attribute*), 344
i_item (*mafw.processor.base.Processor property*), 378
i_item (*mafw.processor.Processor property*), 364
Importer (*class in mafw.processor_library.importer*), 402
ImporterExample (*class in mafw.examples.importer_example*), 326
in_loop_customization() (*mafw.processor_library.abstract_plotter.GenericPlotter method*), 391
in_sync (*mafw.devtools.toolchain.base.ToolCheckResult property*), 291
in_sync (*mafw.devtools.toolchain.ToolCheckResult property*), 283
Init (*mafw.enumerators.ProcessorStatus attribute*), 323
init() (*mafw.db.std_tables.StandardTable class method*), 183
init() (*mafw.db.std_tables.TriggerStatus class method*), 184
init() (*mafw.runner.MAFwApplication method*), 415
initialise_parameters() (*mafw.processor.base.Processor method*), 375
initialise_parameters() (*mafw.processor.Processor method*), 361
initialise_parameters() (*mafw.tools.loop_tuning.LoopTuningMixin method*), 484
InputElement (*class in mafw.examples.importer_example*), 328
InputElementDoesNotExist, 326
insert() (*mafw.processor.processor_list.ProcessorList method*), 386
insert() (*mafw.processor.ProcessorList method*), 366
install_completion() (*in module mafw.tools.click_extensions*), 479
install_exception_handler() (*in module mafw.steering_gui.utils.exception_handling*), 457
INT_RANGE (*in module mafw.steering_gui.utils.constants*), 456
interpret() (*mafw.processor_library.importer.FilenameParser method*), 402
InvalidConfigurationError, 341
InvalidFilterConditionIssue, 341
InvalidFilterLogicIssue, 341
InvalidSteeringFile, 341
invoke() (*mafw.scripts.mafw_exe.MAFwGroup method*), 425
invoke() (*mafw.tools.click_extensions.AbbreviateGroup method*), 477
is_ancestor() (*in module mafw.devtools.documentation*), 227
is_ancestor() (*in module mafw.devtools.git*), 247
is_bound (*mafw.db.db_filter.ModelFilter property*), 167
is_bugged_warning_capture_version() (*in module mafw.scripts.mafw_exe*), 425
is_complete() (*mafw.steering.models.ConditionalFilterConfig method*), 430
is_complete() (*mafw.steering_gui.views.filter_editors.ConditionDefinit method*), 465
is_complete() (*mafw.steering_gui.views.filter_editors.ConditionEditor method*), 465

is_condition_warning() (*mafw.steering_gui.models.filter_tree_model.FilterTreeModel method*), 451
is_data_frame_empty() (*mafw.processor_library.abstract_plotter.GenericPlotter method*), 391
is_data_frame_empty() (*mafw.processor_library.sns_plotter.SNSPlotter method*), 412
is_db_configuration_enabled() (*mafw.steering.builder.SteeringBuilder method*), 427
is_db_configuration_enabled() (*mafw.steering_gui.controllers.steering_controller.SteeringController method*), 439
is_enabled() (*mafw.steering_gui.views.database_editor.DatabaseEditor method*), 462
is_enabled() (*mafw.steering_gui.views.DatabaseEditor method*), 459
is_found (*mafw.processor_library.importer.FilenameElement property*), 401
is_free_threading() (*in module mafw.tools.parallel*), 486
is_group() (*mafw.steering_gui.models.pipeline.PipelineItem method*), 452
is_group() (*mafw.steering_gui.models.PipelineItem method*), 449
is_logic_warning() (*mafw.steering_gui.models.filter_tree_model.FilterTreeModel method*), 451
is_optional (*mafw.processor.parameters.PassiveParameter property*), 383
is_optional (*mafw.processor.PassiveParameter property*), 354
is_optional (*mafw.processor_library.importer.FilenameElement property*), 401
is_output_existing() (*mafw.processor_library.abstract_plotter.GenericPlotter method*), 391
is_prime() (*in module mafw.examples.loop_modifier*), 335
is_script_already_installed() (*in module mafw.tools.click_extensions*), 480
is_set (*mafw.processor.parameters.PassiveParameter property*), 383
is_set (*mafw.processor.PassiveParameter property*), 354
is_valid() (*mafw.steering_gui.views.database_editor.TransportWidget method*), 463
is_valid() (*mafw.steering_gui.views.filter_editors.LogicEditor method*), 466
Issue (*class in mafw.devtools.toolchain*), 278
Issue (*class in mafw.devtools.toolchain.base*), 286
item (*mafw.processor.base.Processor property*), 378
item (*mafw.processor.Processor property*), 364
items() (*mafw.db.model_register.ModelRegister method*), 181
iter_local_mafw_docs_zips() (*in module mafw.devtools.documentation*), 227
iter_local_mafw_docs_zips() (*in module mafw.devtools.documentation.builder*), 237
iter_local_pylock_reference_files() (*in module mafw.devtools.gitlab*), 251
iter_local_pylock_reference_files() (*in module mafw.devtools.gitlab.registry*), 260
iter_specifiers() (*in module mafw.devtools.dependencies.freeze*), 218

J

join_dependent_models() (*mafw.db.db_model.MAFwBaseModel class method*), 172

L

latest (*in module mafw.devtools.cli.dependencies*), 197
LazyImportPlugin (*class in mafw.lazy_import*), 338
LazyImportProcessor (*class in mafw.lazy_import*), 338
LazyImportUserInterface (*class in mafw.lazy_import*), 339
LEGACY_KEYS (*in module mafw.db.db_connection*), 155
list_generic_package_files() (*in module mafw.devtools.gitlab*), 251
list_generic_package_files() (*in module mafw.devtools.gitlab.registry*), 260
list_generic_packages() (*in module mafw.devtools.gitlab*), 251
list_generic_packages() (*in module mafw.devtools.gitlab.registry*), 260
list_groups() (*mafw.steering.builder.SteeringBuilder method*), 427
list_groups() (*mafw.steering_gui.controllers.steering_controller.Steer method*), 439
list_mafw_docs_generic_packages() (*in module mafw.devtools.documentation*), 228
list_mafw_docs_generic_packages() (*in module mafw.devtools.gitlab.docs_registry*), 256
list_processors() (*mafw.steering.builder.SteeringBuilder method*), 427
list_processors() (*mafw.steering_gui.controllers.steering_controller. method*), 439
ListEditor (*class in mafw.steering_gui.views.filter_editors*), 466
ListValueRow (*class in mafw.steering_gui.views.filter_editors*), 466
LMPlot (*class in mafw.processor_library.sns_plotter*), 408
load() (*mafw.steering_gui.controllers.steering_controller.SteeringContr method*), 439
load_db_models_plugins() (*mafw.plugin_manager.MAFwPluginManager method*), 349

load_plugins() (*mafw.plugin_manager.MAFwPluginManager method*), 349
load_plugins() (*mafw.steering_gui.controllers.steering_controller.SteeringController method*), 439
load_processor_plugins() (*mafw.plugin_manager.MAFwPluginManager method*), 349
load_pylock_packages() (*in module mafw.devtools.dependencies*), 203
load_pylock_packages() (*in module mafw.devtools.dependencies.compile*), 213
load_pyproject_doc() (*in module mafw.devtools.dependencies*), 203
load_pyproject_doc() (*in module mafw.devtools.dependencies.compile*), 214
load_steering_file() (*in module mafw.tools.toml_tools*), 491
load_steering_file_legacy() (*in module mafw.tools.toml_tools*), 492
load_user_interface_plugins() (*mafw.plugin_manager.MAFwPluginManager method*), 350
LoadedPlugins (*class in mafw.plugin_manager*), 348
local_resource_acquisition (*mafw.processor.base.Processor property*), 378
local_resource_acquisition (*mafw.processor.Processor property*), 364
logger_setup() (*in module mafw.scripts.mafw_exe*), 426
logic (*mafw.steering.models.FieldFilterConfig property*), 430
logic (*mafw.steering.models.ModelFilterConfig property*), 432
logic_key_for_index() (*mafw.steering_gui.models.filter_tree_model.FilterTreeModel method*), 451
logic_key_for_item() (*mafw.steering_gui.models.filter_tree_model.FilterTreeModel method*), 451
logic_name (*mafw.db.db_filter.ModelFilter attribute*), 167
LogicalNode (*class in mafw.db.db_filter*), 162
LogicalOp (*class in mafw.enumerators*), 321
LogicEditor (*class in mafw.steering_gui.views.filter_editors*), 466
LogicKey (*class in mafw.steering_gui.models.filter_tree_model*), 452
LogicSyntaxHighlighter (*class in mafw.steering_gui.views.filter_editors*), 466
LogicTextEdit (*class in mafw.steering_gui.views.filter_editors*), 466
loop_type (*mafw.processor.base.Processor attribute*), 378
loop_type (*mafw.processor.Processor attribute*), 364
loop_values (*mafw.models.loop_payloads.LoopResult attribute*), 346
looping_status (*mafw.models.LoopResult attribute*), 344
looping_status (*mafw.processor.base.Processor property*), 378
looping_status (*mafw.processor.Processor property*), 364
LoopingStatus (*class in mafw.enumerators*), 322
LoopItem (*class in mafw.models*), 344
LoopItem (*class in mafw.models.loop_payloads*), 346
LoopResult (*class in mafw.models*), 344
LoopResult (*class in mafw.models.loop_payloads*), 346
LoopTuningMixin (*class in mafw.tools.loop_tuning*), 484
LoopType (*class in mafw.enumerators*), 321
LOWER_BOUND_RE (*in module mafw.devtools.toolchain.base*), 291

M

mafw
    module, 151
mafw command line option
    -D, 417
    --debug, 417
    --log-level, 417
    --no-banner, 417
    --ui, 417
    --version, 417
    -v, 417
mafw.active
    module, 152
mafw.db
    module, 153
mafw.db.db_configurations
    module, 154
mafw.db.db_connection
    module, 154
mafw.db.db_filter
    module, 155
mafw.db.db_model
    module, 170
mafw.db.db_types
    module, 176
mafw.db.db_wizard
    module, 176
mafw.db.fields
    module, 177
mafw.db.model_register
    module, 179
mafw.db.std_tables
    module, 182
mafw.db.trigger
    module, 185
mafw.decorators

module, 190
mafw.devtools
module, 194
mafw.devtools.cli
module, 196
mafw.devtools.cli.cli
module, 196
mafw.devtools.cli.completion
module, 196
mafw.devtools.cli.dependencies
module, 196
mafw.devtools.cli.documentation
module, 197
mafw.devtools.cli.release
module, 198
mafw.devtools.cli.toolchain
module, 198
mafw.devtools.cli.toolchain.bootstrap_cmd
module, 199
mafw.devtools.cli.toolchain.check_cmd
module, 199
mafw.devtools.cli.toolchain.list_cmd
module, 199
mafw.devtools.cli.toolchain.update_cmd
module, 199
mafw.devtools.cli.toolchain.verify_cmd
module, 199
mafw.devtools.dependencies
module, 200
mafw.devtools.dependencies.audit
module, 208
mafw.devtools.dependencies.compare
module, 209
mafw.devtools.dependencies.compile
module, 212
mafw.devtools.dependencies.freeze
module, 215
mafw.devtools.dependencies.verify
module, 220
mafw.devtools.documentation
module, 220
mafw.devtools.documentation.builder
module, 233
mafw.devtools.documentation.requirements
module, 239
mafw.devtools.documentation.server
module, 240
mafw.devtools.documentation.versions
module, 243
mafw.devtools.git
module, 246
mafw.devtools.gitlab
module, 248
mafw.devtools.gitlab.api
module, 253
mafw.devtools.gitlab.docs_registry
module, 254
mafw.devtools.gitlab.registry
module, 257
mafw.devtools.release
module, 262
mafw.devtools.release.changelog
module, 269
mafw.devtools.release.checks
module, 270
mafw.devtools.release.notes
module, 271
mafw.devtools.release.versioning
module, 272
mafw.devtools.toolchain
module, 277
mafw.devtools.toolchain.base
module, 284
mafw.devtools.toolchain.default_registry
module, 291
mafw.devtools.toolchain.filtering
module, 292
mafw.devtools.toolchain.precommit_modifier
module, 293
mafw.devtools.toolchain.pypi
module, 294
mafw.devtools.toolchain.pyproject_modifier
module, 295
mafw.devtools.toolchain.registry
module, 298
mafw.devtools.toolchain.tools
module, 299
mafw.devtools.toolchain.tools.git_cliff
module, 307
mafw.devtools.toolchain.tools.hatch
module, 308
mafw.devtools.toolchain.tools.mypy
module, 309
mafw.devtools.toolchain.tools.pip_audit
module, 310
mafw.devtools.toolchain.tools.precommit
module, 311
mafw.devtools.toolchain.tools.pytest
module, 312
mafw.devtools.toolchain.tools.pyupgrade
module, 314
mafw.devtools.toolchain.tools.ruff
module, 316
mafw.devtools.toolchain.tools.sphinx
module, 318
mafw.devtools.toolchain.tools.uv
module, 320
mafw.enumerators
module, 321
mafw.examples
module, 323
mafw.examples.db_processors
module, 324
mafw.examples.importer_example
module, 326
mafw.examples.loop_modifier

module, 329
mafw.examples.multi_primary
module, 335
mafw.examples.processor_list
module, 335
mafw.examples.sum_processor
module, 336
mafw.hookspecs
module, 337
mafw.lazy_import
module, 337
mafw.mafw_errors
module, 340
mafw.models
module, 344
mafw.models.filter_schema
module, 345
mafw.models.loop_payloads
module, 345
mafw.models.parameter_schema
module, 346
mafw.models.processor_schema
module, 347
mafw.plugin_manager
module, 347
mafw.plugins
module, 351
mafw.processor
module, 351
mafw.processor.base
module, 368
mafw.processor.meta
module, 379
mafw.processor.parameters
module, 380
mafw.processor.processor_list
module, 385
mafw.processor.utils
module, 387
mafw.processor_library
module, 387
mafw.processor_library.abstract_plotter
module, 387
mafw.processor_library.db_init
module, 392
mafw.processor_library.importer
module, 399
mafw.processor_library.sns_plotter
module, 404
mafw.runner
module, 414
mafw.scripts
module, 416
mafw.scripts.mafw_exe
module, 416
mafw.steering
module, 426
mafw.steering.builder
module, 426
mafw.steering.models
module, 429
mafw.steering.serializer
module, 434
mafw.steering.validation
module, 434
mafw.steering_gui
module, 434
mafw.steering_gui.actions
module, 435
mafw.steering_gui.app
module, 435
mafw.steering_gui.controllers
module, 436
mafw.steering_gui.controllers.steering_controller
module, 436
mafw.steering_gui.dialogs
module, 442
mafw.steering_gui.dialogs.about_dialog
module, 443
mafw.steering_gui.dialogs.add_field_filter_dialog
module, 443
mafw.steering_gui.dialogs.add_model_dialog
module, 444
mafw.steering_gui.dialogs.add_processor_dialog
module, 444
mafw.steering_gui.dialogs.steering_text_editor
module, 444
mafw.steering_gui.dialogs.validation_issues_dialog
module, 445
mafw.steering_gui.main_window
module, 445
mafw.steering_gui.models
module, 449
mafw.steering_gui.models.filter_tree_model
module, 450
mafw.steering_gui.models.pipeline
module, 452
mafw.steering_gui.models.processor_pipeline_model
module, 453
mafw.steering_gui.models.steering_tree_model
module, 454
mafw.steering_gui.threading
module, 454
mafw.steering_gui.threading.workers
module, 455
mafw.steering_gui.utils
module, 456
mafw.steering_gui.utils.constants
module, 456
mafw.steering_gui.utils.exception_handling
module, 456
mafw.steering_gui.utils.gui_helpers
module, 458
mafw.steering_gui.views
module, 458
mafw.steering_gui.views.database_editor

module, 461
mafw.steering_gui.views.filter_editors
module, 464
mafw.steering_gui.views.filter_manager
module, 468
mafw.steering_gui.views.filter_tree
module, 469
mafw.steering_gui.views.globals_editor
module, 470
mafw.steering_gui.views.group_editor
module, 470
mafw.steering_gui.views.meta_editor
module, 471
mafw.steering_gui.views.pipeline_editor
module, 471
mafw.steering_gui.views.processor_editor
module, 472
mafw.steering_gui.views.processor_parameter_editor
module, 472
mafw.steering_gui.views.steering_tree
module, 473
mafw.steering_gui.views.ui_editor
module, 474
mafw.timer
module, 474
mafw.tools
module, 476
mafw.tools.click_extensions
module, 476
mafw.tools.db_tools
module, 480
mafw.tools.file_tools
module, 482
mafw.tools.generics
module, 483
mafw.tools.loop_tuning
module, 484
mafw.tools.pandas_tools
module, 485
mafw.tools.parallel
module, 486
mafw.tools.regexp
module, 486
mafw.tools.shell_tools
module, 487
mafw.tools.toml_tools
module, 489
mafw.ui
module, 493
mafw.ui.abstract_user_interface
module, 493
mafw.ui.console_user_interface
module, 495
mafw.ui.rich_user_interface
module, 497
mafw_hookimpl (*in module mafw*), 151
mafw_model_register (*in module mafw.db.db_model*), 175
mafw-completion-install command line option
-F, 418
--force, 418
--shell, 418
-s, 418
mafw-completion-show command line option
--shell, 419
-s, 419
mafw-db-wizard command line option
--engine, 422
--host, 422
--ignore-unknown, 422
--no-ignore-unknown, 422
--no-overwrite, 422
--no-preserve-order, 422
--no-snake-case, 422
--output-file, 421
--overwrite, 422
--password, 422
--port, 422
--preserve-order, 422
--schema, 421
--snake-case, 422
--tables, 421
--user, 422
--username, 422
--with-views, 422
--without-views, 422
-e, 422
-o, 421
-p, 422
-s, 421
-t, 421
-u, 422
DATABASE, 422
mafw-run command line option
STEERING_FILE, 423
mafw-steering command line option
--db-engine, 424
--db-url, 424
--ext-plugins, 424
--no-ext-plugin, 424
--no-open-editor, 424
--no-show, 424
--open-editor, 424
--show, 424
STEERING_FILE, 424
MAFwApplication (*class in mafw.runner*), 414
MAFwBaseModel (*class in mafw.db.db_model*), 170
MAFwBaseModelDoesNotExist, 170
MAFwException, 341
MAFwGroup (*class in mafw.scripts.mafw_exe*), 424
MAFwPluginManager (*class in mafw.plugin_manager*), 349
main() (*in module mafw.steering_gui.app*), 435
main() (*mafw.scripts.mafw_exe.MAFwGroup method*), 425

MainWindow (*class in mafw.steering_gui.main_window*), 446
make_kv() (*in module mafw.tools.db_tools*), 482
make_prefixed_suffixed_name() (*in module mafw.db.db_model*), 175
MASTER_RE (*in module mafw.db.db_filter*), 169
matches() (*mafw.steering_gui.views.filter_editors.FreeNameWidget method*), 466
matplotlib_backend (*mafw.processor_library.sns_plotter.SNSPlotter attribute*), 413
MAX_ENV_RESOLUTION_PASSES (*in module mafw.tools.toml_tools*), 493
max_loading_delay (*mafw.plugin_manager.MAFwPluginManager attribute*), 350
max_workers (*mafw.processor.base.Processor attribute*), 378
max_workers (*mafw.processor.Processor attribute*), 364
MetadataEditor (*class in mafw.steering_gui.views*), 459
MetadataEditor (*class in mafw.steering_gui.views.meta_editor*), 471
method_combo (*mafw.steering_gui.views.database_editor.AuthenticationWidget property*), 462
mirror_version() (*in module mafw.devtools.documentation*), 228
mirror_version() (*in module mafw.devtools.documentation.versions*), 244
MissingAttribute, 342
MissingDatabase, 342
MissingOptionalDependency, 342
MissingOverloadedMethod, 342
MissingProcessorsToRunIssue, 342
MissingSQLStatement, 342
MissingSuperCall, 342
MissingTokenError, 157
MissingUserInterfaceSectionIssue, 342
ModelError, 342
ModelFieldsTableModel (*class in mafw.steering_gui.views.filter_editors*), 466
ModelFilter (*class in mafw.db.db_filter*), 162
ModelFilterConfig (*class in mafw.steering.models*), 431
ModelFilterEditor (*class in mafw.steering_gui.views.filter_editors*), 467
ModelInfo (*class in mafw.steering.models*), 432
ModelRegister (*class in mafw.db.model_register*), 180
ModifyLoopProcessor (*class in mafw.examples.loop_modifier*), 334
module
    mafw, 151
    mafw.active, 152
    mafw.db, 153
    mafw.db.db_configurations, 154
    mafw.db.db_connection, 154
    mafw.db.db_filter, 155
    mafw.db.db_model, 170
    mafw.db.db_types, 176
    mafw.db.db_wizard, 176
    mafw.db.fields, 177
    mafw.db.model_register, 179
    mafw.db.std_tables, 182
    mafw.db.trigger, 185
    mafw.decorators, 190
    mafw.devtools, 194
    mafw.devtools.cli, 196
    mafw.devtools.cli.cli, 196
    mafw.devtools.cli.completion, 196
    mafw.devtools.cli.dependencies, 196
    mafw.devtools.cli.documentation, 197
    mafw.devtools.cli.release, 198
    mafw.devtools.cli.toolchain, 198
    mafw.devtools.cli.toolchain.bootstrap_cmd, 199
    mafw.devtools.cli.toolchain.check_cmd, 199
    mafw.devtools.cli.toolchain.list_cmd, 199
    mafw.devtools.cli.toolchain.update_cmd, 199
    mafw.devtools.cli.toolchain.verify_cmd, 199
    mafw.devtools.dependencies, 200
    mafw.devtools.dependencies.audit, 208
    mafw.devtools.dependencies.compare, 209
    mafw.devtools.dependencies.compile, 212
    mafw.devtools.dependencies.freeze, 215
    mafw.devtools.dependencies.verify, 220
    mafw.devtools.documentation, 220
    mafw.devtools.documentation.builder, 233
    mafw.devtools.documentation.requirements, 239
    mafw.devtools.documentation.server, 240
    mafw.devtools.documentation.versions, 243
    mafw.devtools.git, 246
    mafw.devtools.gitlab, 248
    mafw.devtools.gitlab.api, 253
    mafw.devtools.gitlab.docs_registry, 254
    mafw.devtools.gitlab.registry, 257
    mafw.devtools.release, 262
    mafw.devtools.release.changelog, 269
    mafw.devtools.release.checks, 270
    mafw.devtools.release.notes, 271
    mafw.devtools.release.versioning, 272
    mafw.devtools.toolchain, 277
    mafw.devtools.toolchain.base, 284
    mafw.devtools.toolchain.default_registry, 291

mafw.devtools.toolchain.filtering, 292
mafw.devtools.toolchain.precommit_modifier, 293
mafw.devtools.toolchain.pypi, 294
mafw.devtools.toolchain.pyproject_modifier, 295
mafw.devtools.toolchain.registry, 298
mafw.devtools.toolchain.tools, 299
mafw.devtools.toolchain.tools.git_cliff, 307
mafw.devtools.toolchain.tools.hatch, 308
mafw.devtools.toolchain.tools.mypy, 309
mafw.devtools.toolchain.tools.pip_audit, 310
mafw.devtools.toolchain.tools.precommit, 311
mafw.devtools.toolchain.tools.pytest, 312
mafw.devtools.toolchain.tools.pyupgrade, 314
mafw.devtools.toolchain.tools.ruff, 316
mafw.devtools.toolchain.tools.sphinx, 318
mafw.devtools.toolchain.tools.uv, 320
mafw.enumerators, 321
mafw.examples, 323
mafw.examples.db_processors, 324
mafw.examples.importer_example, 326
mafw.examples.loop_modifier, 329
mafw.examples.multi_primary, 335
mafw.examples.processor_list, 335
mafw.examples.sum_processor, 336
mafw.hookspecs, 337
mafw.lazy_import, 337
mafw.mafw_errors, 340
mafw.models, 344
mafw.models.filter_schema, 345
mafw.models.loop_payloads, 345
mafw.models.parameter_schema, 346
mafw.models.processor_schema, 347
mafw.plugin_manager, 347
mafw.plugins, 351
mafw.processor, 351
mafw.processor.base, 368
mafw.processor.meta, 379
mafw.processor.parameters, 380
mafw.processor.processor_list, 385
mafw.processor.utils, 387
mafw.processor_library, 387
mafw.processor_library.abstract_plotter, 387
mafw.processor_library.db_init, 392
mafw.processor_library.importer, 399
mafw.processor_library.sns_plotter, 404
mafw.runner, 414
mafw.scripts, 416
mafw.scripts.mafw_exe, 416
mafw.steering, 426
mafw.steering.builder, 426
mafw.steering.models, 429
mafw.steering.serializer, 434
mafw.steering.validation, 434
mafw.steering_gui, 434
mafw.steering_gui.actions, 435
mafw.steering_gui.app, 435
mafw.steering_gui.controllers, 436
mafw.steering_gui.controllers.steering_controller, 436
mafw.steering_gui.dialogs, 442
mafw.steering_gui.dialogs.about_dialog, 443
mafw.steering_gui.dialogs.add_field_filter_dialog, 443
mafw.steering_gui.dialogs.add_model_dialog, 444
mafw.steering_gui.dialogs.add_processor_dialog, 444
mafw.steering_gui.dialogs.steering_text_editor, 444
mafw.steering_gui.dialogs.validation_issues_dialog, 445
mafw.steering_gui.main_window, 445
mafw.steering_gui.models, 449
mafw.steering_gui.models.filter_tree_model, 450
mafw.steering_gui.models.pipeline, 452
mafw.steering_gui.models.processor_pipeline_model, 453
mafw.steering_gui.models.steering_tree_model, 454
mafw.steering_gui.threading, 454
mafw.steering_gui.threading.workers, 455
mafw.steering_gui.utils, 456
mafw.steering_gui.utils.constants, 456
mafw.steering_gui.utils.exception_handling, 456
mafw.steering_gui.utils.gui_helpers, 458
mafw.steering_gui.views, 458
mafw.steering_gui.views.database_editor, 461
mafw.steering_gui.views.filter_editors, 464
mafw.steering_gui.views.filter_manager, 468
mafw.steering_gui.views.filter_tree, 469
mafw.steering_gui.views.globals_editor, 470
mafw.steering_gui.views.group_editor, 470
mafw.steering_gui.views.meta_editor, 471
mafw.steering_gui.views.pipeline_editor,

471
mafw.steering_gui.views.processor_editor, 472
mafw.steering_gui.views.processor_parameter_editor, 472
mafw.steering_gui.views.steering_tree, 473
mafw.steering_gui.views.ui_editor, 474
mafw.timer, 474
mafw.tools, 476
mafw.tools.click_extensions, 476
mafw.tools.db_tools, 480
mafw.tools.file_tools, 482
mafw.tools.generics, 483
mafw.tools.loop_tuning, 484
mafw.tools.pandas_tools, 485
mafw.tools.parallel, 486
mafw.tools.regexp, 486
mafw.tools.shell_tools, 487
mafw.tools.toml_tools, 489
mafw.ui, 493
mafw.ui.abstract_user_interface, 493
mafw.ui.console_user_interface, 495
mafw.ui.rich_user_interface, 497
move_pipeline_entry() (*mafw.steering_gui.controllers.steering_controller.SteeringController method*), 439
move_processor_in_run_list() (*mafw.steering_gui.controllers.steering_controller.SteeringController method*), 439
MypyTool (*class in mafw.devtools.toolchain.tools*), 300
MypyTool (*class in mafw.devtools.toolchain.tools.mypy*), 309
MySQLDialect (*class in mafw.db.trigger*), 185

N

n_item (*mafw.models.loop_payloads.LoopItem attribute*), 346
n_item (*mafw.models.loop_payloads.LoopResult attribute*), 346
n_item (*mafw.models.LoopItem attribute*), 344
n_item (*mafw.models.LoopResult attribute*), 344
n_item (*mafw.processor.base.Processor property*), 378
n_item (*mafw.processor.Processor property*), 364
name (*mafw.devtools.toolchain.base.HostTool property*), 286
name (*mafw.devtools.toolchain.base.ProjectTool property*), 289
name (*mafw.devtools.toolchain.base.ToolChainTool property*), 290
name (*mafw.devtools.toolchain.HostTool property*), 278
name (*mafw.devtools.toolchain.ProjectTool property*), 281
name (*mafw.devtools.toolchain.ToolChainTool property*), 282
name (*mafw.processor.base.Processor attribute*), 378
name (*mafw.processor.Processor attribute*), 364
name (*mafw.processor.processor_list.ProcessorList property*), 386
name (*mafw.processor.ProcessorList property*), 367
name (*mafw.processor_library.importer.FilenameElement property*), 401
name (*mafw.runner.MAFwApplication attribute*), 416
name (*mafw.ui.abstract_user_interface.UserInterfaceBase attribute*), 495
name (*mafw.ui.console_user_interface.ConsoleInterface attribute*), 497
name (*mafw.ui.rich_user_interface.RichInterface attribute*), 499
name() (*mafw.steering_gui.models.pipeline.PipelineItem method*), 452
name() (*mafw.steering_gui.models.PipelineItem method*), 449
NameNode (*in module mafw.db.db_filter*), 169
nested_list (*mafw.processor.processor_list.ProcessorList attribute*), 386
nested_list (*mafw.processor.ProcessorList attribute*), 367
new_defaults (*mafw.processor.base.Processor attribute*), 378
new_defaults (*mafw.processor.Processor attribute*), 365
new_entry (*mafw.db.db_filter.ProcessorFilter property*), 168
next_minor_version() (*in module mafw.devtools.release.versioning*), 275
normalize_dependency_registry_item() (*in module mafw.devtools.gitlab*), 251
normalize_dependency_registry_item() (*in module mafw.devtools.gitlab.registry*), 260
normalize_hatch_segments() (*in module mafw.devtools.release*), 266
normalize_hatch_segments() (*in module mafw.devtools.release.versioning*), 275
normalize_mafw_version() (*in module mafw.devtools.gitlab*), 252
normalize_mafw_version() (*in module mafw.devtools.gitlab.registry*), 261
normalize_registry_item() (*in module mafw.devtools.documentation*), 228
normalize_registry_item() (*in module mafw.devtools.documentation.builder*), 237
normalize_sql_spaces() (*in module mafw.tools.regexp*), 487
NOTICE_FILE (*in module mafw.devtools.release.versioning*), 276
NOTICE_VERSION_PATTERN (*in module mafw.devtools.release.versioning*), 276
NotNode (*in module mafw.db.db_filter*), 169

O

OK (*mafw.scripts.mafw_exe.ReturnValue attribute*), 425
OLD_VERSION_TO_BE_PATCHED (*in module mafw.devtools.documentation.builder*),

239
oldest (*in module mafw.devtools.cli.dependencies*), 197
on_looping_status_set() (*mafw.processor.base.Processor method*), 375
on_looping_status_set() (*mafw.processor.Processor method*), 362
on_processor_status_change() (*mafw.processor.base.Processor method*), 376
on_processor_status_change() (*mafw.processor.Processor method*), 362
or_() (*in module mafw.db.trigger*), 190
orphan_protector() (*in module mafw.decorators*), 193
OrphanFile (*class in mafw.db.std_tables*), 183
OrphanFileDoesNotExist, 182
output_filename_list (*mafw.processor_library.sns_plotter.SNSPlotter attribute*), 413

P

P (*class in mafw.decorators*), 191
P (*in module mafw.steering_gui.controllers.steering_controller*), 442
package_name (*mafw.devtools.toolchain.base.ProjectTool property*), 289
package_name (*mafw.devtools.toolchain.ProjectTool property*), 281
package_name (*mafw.devtools.toolchain.tools.git_cliff.GitCliffTool property*), 308
package_name (*mafw.devtools.toolchain.tools.GitCliffTool property*), 299
package_name (*mafw.devtools.toolchain.tools.mypy.MypyTool property*), 310
package_name (*mafw.devtools.toolchain.tools.MypyTool property*), 301
package_name (*mafw.devtools.toolchain.tools.pip_audit.PipAuditTool property*), 311
package_name (*mafw.devtools.toolchain.tools.PipAuditTool property*), 301
package_name (*mafw.devtools.toolchain.tools.precommit.PreCommitTool property*), 312
package_name (*mafw.devtools.toolchain.tools.PreCommitTool property*), 302
package_name (*mafw.devtools.toolchain.tools.pytest.PytestTool property*), 313
package_name (*mafw.devtools.toolchain.tools.PytestTool property*), 303
package_name (*mafw.devtools.toolchain.tools.pyupgrade.PyUpgradeTool property*), 316
package_name (*mafw.devtools.toolchain.tools.ruff.RuffTool property*), 318
package_name (*mafw.devtools.toolchain.tools.RuffTool property*), 305
package_name (*mafw.devtools.toolchain.tools.sphinx.SphinxTool property*), 319
package_name (*mafw.devtools.toolchain.tools.SphinxTool property*), 306
PackageChange (*class in mafw.devtools.dependencies*), 200
PackageChange (*class in mafw.devtools.dependencies.compare*), 209
parallel_for_loop() (*in module mafw.decorators*), 193
parallel_for_loop_with_queue() (*in module mafw.decorators*), 193
ParallelForLoop (*mafw.enumerators.LoopType attribute*), 322
ParallelForLoopWithQueue (*mafw.enumerators.LoopType attribute*), 322
parameter_schema() (*mafw.lazy_import.ProcessorClassProtocol method*), 339
parameter_schema() (*mafw.processor.base.Processor class method*), 369
parameter_schema() (*mafw.processor.Processor class method*), 355
ParameterConfig (*class in mafw.steering.models*), 432
ParameterDelegate (*class in mafw.steering_gui.views.processor_parameter_editor*), 472
ParameterSchema (*class in mafw.models*), 344
ParameterSchema (*class in mafw.models.parameter_schema*), 346
ParameterSchemaStatus (*class in mafw.steering.models*), 432
ParameterSource (*class in mafw.steering.models*), 432
ParametersWidget (*class in mafw.steering_gui.views.database_editor*), 463
ParameterTableModel (*class in mafw.steering_gui.views.processor_parameter_editor*), 472
ParameterType (*class in mafw.processor.parameters*), 382
parse() (*mafw.db.db_filter.ExprParser method*), 160
parse_and() (*mafw.db.db_filter.ExprParser method*), 161
parse_args() (*mafw.scripts.mafw_exe.MAFwGroup method*), 425
parse_atom() (*mafw.db.db_filter.ExprParser method*), 161
parse_mafw_docs_zip_filename() (*in module mafw.devtools.documentation*), 228
parse_mafw_docs_zip_filename() (*in module mafw.devtools.documentation.builder*), 237
parse_not() (*mafw.db.db_filter.ExprParser method*), 161
parse_or() (*mafw.db.db_filter.ExprParser method*),

161
parse_processor_name() (*in module mafw.tools.regexp*), 487
parse_pylock_reference_filename() (*in module mafw.devtools.gitlab*), 252
parse_pylock_reference_filename() (*in module mafw.devtools.gitlab.registry*), 261
parse_python_version() (*in module mafw.devtools.dependencies*), 204
parse_python_version() (*in module mafw.devtools.dependencies.compile*), 214
parse_sphinx_log() (*in module mafw.devtools.documentation*), 229
parse_sphinx_log() (*in module mafw.devtools.documentation.builder*), 238
parse_stable_tag() (*in module mafw.devtools.release*), 267
parse_stable_tag() (*in module mafw.devtools.release.checks*), 271
parse_version() (*in module mafw.devtools.release*), 267
parse_version() (*in module mafw.devtools.release.versioning*), 275
parse_version_tuple() (*in module mafw.devtools.documentation*), 229
parse_version_tuple() (*in module mafw.devtools.documentation.builder*), 238
ParseError, 157
ParserConfigurationError, 342
ParsingError, 342
PassiveParameter (*class in mafw.processor*), 353
PassiveParameter (*class in mafw.processor.parameters*), 382
password_edit (*mafw.steering_gui.views.database_editor.AuthenticationWidget property*), 462
patch_data_frame() (*mafw.processor_library.abstract_plotter.DataRetriever method*), 388
patch_data_frame() (*mafw.processor_library.abstract_plotter.GenericPlotter method*), 392
patch_data_frame() (*mafw.processor_library.sns_plotter.HDFPdDataRetriever method*), 408
patch_data_frame() (*mafw.processor_library.sns_plotter.PdDataRetriever method*), 409
path_encoder() (*in module mafw.tools.toml_tools*), 492
PathEditor (*class in mafw.steering_gui.views.processor_parameter_editor*), 473
PathItem (*class in mafw.tools.toml_tools*), 489
pattern (*mafw.processor_library.importer.FilenameElement property*), 401
payload (*mafw.models.loop_payloads.LoopItem attribute*), 346
payload (*mafw.models.loop_payloads.LoopResult attribute*), 346
payload (*mafw.models.LoopItem attribute*), 344
payload (*mafw.models.LoopResult attribute*), 344
PdDataRetriever (*class in mafw.processor_library.sns_plotter*), 408
peek() (*mafw.db.db_filter.ExprParser method*), 161
PeeweeModelWithMeta (*class in mafw.db.db_types*), 176
PipAuditTool (*class in mafw.devtools.toolchain.tools*), 301
PipAuditTool (*class in mafw.devtools.toolchain.tools.pip_audit*), 310
pipeline() (*mafw.steering_gui.models.processor_pipeline_model.Proce method*), 453
pipeline() (*mafw.steering_gui.models.ProcessorPipelineModel method*), 449
pipeline_item_from_index() (*mafw.steering_gui.models.steering_tree_model.SteeringTreeM method*), 454
pipeline_item_from_index() (*mafw.steering_gui.models.SteeringTreeModel method*), 449
PipelineEditor (*class in mafw.steering_gui.views*), 460
PipelineEditor (*class in mafw.steering_gui.views.pipeline_editor*), 471
PipelineItem (*class in mafw.steering_gui.models*), 449
PipelineItem (*class in mafw.steering_gui.models.pipeline*), 452
PipelineRoles (*class in mafw.steering_gui.models.processor_pipeline_model*), 453
pipx_package_name (*mafw.devtools.toolchain.base.HostTool property*), 286
pipx_package_name (*mafw.devtools.toolchain.HostTool property*), 278
pipx_package_name (*mafw.devtools.toolchain.tools.hatch.HatchTool property*), 308
pipx_package_name (*mafw.devtools.toolchain.tools.HatchTool property*), 300
pipx_package_name (*mafw.devtools.toolchain.tools.uv.UvTool property*), 321
pipx_package_name (*mafw.devtools.toolchain.tools.UvTool property*), 307
plot() (*mafw.processor_library.abstract_plotter.GenericPlotter method*), 392
plot() (*mafw.processor_library.sns_plotter.CatPlot method*), 406
plot() (*mafw.processor_library.sns_plotter.DisPlot method*), 407
plot() (*mafw.processor_library.sns_plotter.LMPlot method*), 408

plot() (*mafw.processor_library.sns_plotter.RelPlot method*), 410
PlotterMeta (*class in mafw.processor_library.abstract_plotter*), 392
PlotterMixinNotInitialized, 342
PlotterOutput (*class in mafw.db.std_tables*), 183
PlotterOutputDoesNotExist, 182
plugin_manager (*mafw.runner.MAFwApplication attribute*), 416
PluginTypes (*in module mafw.plugin_manager*), 351
post_update() (*mafw.devtools.toolchain.base.ToolChainTool method*), 290
post_update() (*mafw.devtools.toolchain.ToolChainTool method*), 282
post_update() (*mafw.devtools.toolchain.tools.mypy.MypyTool method*), 309
post_update() (*mafw.devtools.toolchain.tools.MypyTool method*), 300
post_update() (*mafw.devtools.toolchain.tools.precommit.PreCommitTool method*), 311
post_update() (*mafw.devtools.toolchain.tools.PreCommitTool method*), 302
post_update() (*mafw.devtools.toolchain.tools.pytest.PytestTool method*), 313
post_update() (*mafw.devtools.toolchain.tools.PytestTool method*), 303
post_update() (*mafw.devtools.toolchain.tools.pyupgrade.PyUpgradeTool method*), 315
post_update() (*mafw.devtools.toolchain.tools.ruff.RuffTool method*), 317
post_update() (*mafw.devtools.toolchain.tools.RuffTool method*), 304
post_update() (*mafw.devtools.toolchain.tools.sphinx.SphinxTool method*), 319
post_update() (*mafw.devtools.toolchain.tools.SphinxTool method*), 305
PostgreSQLDialect (*class in mafw.db.trigger*), 185
PreCommitModifier (*class in mafw.devtools.toolchain.precommit_modifier*), 293
PreCommitTool (*class in mafw.devtools.toolchain.tools*), 301
PreCommitTool (*class in mafw.devtools.toolchain.tools.precommit*), 311
pretty_format_duration() (*in module mafw.timer*), 475
prevent_duplicate_tag() (*in module mafw.devtools.git*), 248
prevent_duplicate_tag() (*in module mafw.devtools.release*), 267
prime_num_found (*mafw.examples.loop_modifier.FindNPrimeNumber attribute*), 330
prime_num_found (*mafw.examples.loop_modifier.FindPrimeNumberInRange attribute*), 332
prime_num_found (*mafw.examples.loop_modifier.FindPrimeNumberInRangePar attribute*), 332
prime_num_found (*mafw.examples.loop_modifier.FindPrimeNumberInR attribute*), 333
prime_num_found (*mafw.examples.loop_modifier.FindPrimeNumberInR attribute*), 334
print_banner() (*in module mafw.scripts.mafw_exe*), 426
print_process_statistics() (*mafw.processor.base.Processor method*), 376
print_process_statistics() (*mafw.processor.Processor method*), 362
process() (*mafw.examples.db_processors.CountStandardTables method*), 325
process() (*mafw.examples.db_processors.FillFileTableProcessor method*), 326
process() (*mafw.examples.importer_example.ImporterExample method*), 328
process() (*mafw.examples.loop_modifier.FindNPrimeNumber method*), 330
process() (*mafw.examples.loop_modifier.FindPrimeNumberInRange method*), 331
process() (*mafw.examples.loop_modifier.FindPrimeNumberInRangePar method*), 332
process() (*mafw.examples.loop_modifier.FindPrimeNumberInRangePar method*), 333
process() (*mafw.examples.loop_modifier.FindPrimeNumberInRangeTur method*), 334
process() (*mafw.examples.loop_modifier.ModifyLoopProcessor method*), 335
process() (*mafw.examples.sum_processor.AccumulatorProcessor method*), 336
process() (*mafw.examples.sum_processor.GaussAdder method*), 337
process() (*mafw.processor.base.Processor method*), 376
process() (*mafw.processor.Processor method*), 362
process() (*mafw.processor_library.abstract_plotter.GenericPlotter method*), 392
process() (*mafw.processor_library.db_init.SQLScriptRunner method*), 394
process() (*mafw.processor_library.db_init.TableCreator method*), 396
process() (*mafw.processor_library.db_init.TriggerRefresher method*), 399
process() (*mafw.processor_library.sns_plotter.SNSPlotter method*), 412
process_matches_http_server() (*in module mafw.devtools.documentation*), 229
process_matches_http_server() (*in module mafw.devtools.documentation.server*), 242
process_response() (*mafw.devtools.release.DocTargetVersionPrompt method*), 263
process_response() (*mafw.devtools.release.versioning.DocTargetVersionPrompt method*), 273
ProcessorInRunningState, 354
Processor (*class in mafw.processor.base*), 368

processor_depends_on_optional() (*in module mafw.decorators*), 193
processor_dict (*mafw.plugin_manager.LoadedPlugins attribute*), 348
processor_exit_status (*mafw.processor.base.Processor attribute*), 378
processor_exit_status (*mafw.processor.Processor attribute*), 365
processor_exit_status (*mafw.processor.processor_list.ProcessorList property*), 386
processor_exit_status (*mafw.processor.ProcessorList property*), 367
processor_list (*mafw.plugin_manager.LoadedPlugins attribute*), 348
processor_schema() (*mafw.lazy_import.ProcessorClassProtocol method*), 339
processor_schema() (*mafw.processor.base.Processor class method*), 369
processor_schema() (*mafw.processor.Processor class method*), 355
processor_status (*mafw.processor.base.Processor attribute*), 378
processor_status (*mafw.processor.Processor attribute*), 365
processor_validator() (*in module mafw.tools.toml_tools*), 492
ProcessorClassProtocol (*class in mafw.lazy_import*), 339
ProcessorConfig (*class in mafw.steering.models*), 432
ProcessorEditor (*class in mafw.steering_gui.views*), 460
ProcessorEditor (*class in mafw.steering_gui.views.processor_editor*), 472
ProcessorExitStatus (*class in mafw.enumerators*), 323
ProcessorFilter (*class in mafw.db.db_filter*), 167
ProcessorFilterEditor (*class in mafw.steering_gui.views.filter_editors*), 467
ProcessorList (*class in mafw.processor*), 365
ProcessorList (*class in mafw.processor.processor_list*), 385
ProcessorMeta (*class in mafw.processor*), 367
ProcessorMeta (*class in mafw.processor.meta*), 379
ProcessorModelsTableModel (*class in mafw.steering_gui.views.filter_editors*), 467
ProcessorParameterEditor (*class in mafw.steering_gui.views*), 460
ProcessorParameterEditor (*class in mafw.steering_gui.views.processor_parameter_editor*), 473
ProcessorParameterError, 343
ProcessorPipeline (*class in mafw.steering_gui.models*), 449
ProcessorPipeline (*class in mafw.steering_gui.models.pipeline*), 453
ProcessorPipelineModel (*class in mafw.steering_gui.models*), 449
ProcessorPipelineModel (*class in mafw.steering_gui.models.processor_pipeline_model*), 453
ProcessorRef (*class in mafw.steering.models*), 433
ProcessorSchema (*class in mafw.models*), 345
ProcessorSchema (*class in mafw.models.processor_schema*), 347
ProcessorSchemaStatus (*class in mafw.steering.models*), 433
ProcessorStatus (*class in mafw.enumerators*), 323
ProcessorsToRunNotListIssue, 343
progress_message (*mafw.processor.base.Processor attribute*), 378
progress_message (*mafw.processor.Processor attribute*), 365
project_python_versions() (*in module mafw.devtools.dependencies*), 204
project_python_versions() (*in module mafw.devtools.dependencies.compile*), 214
project_python_versions_from_doc() (*in module mafw.devtools.dependencies*), 204
project_python_versions_from_doc() (*in module mafw.devtools.dependencies.compile*), 214
ProjectTool (*class in mafw.devtools.toolchain*), 279
ProjectTool (*class in mafw.devtools.toolchain.base*), 286
prompt_question() (*mafw.ui.abstract_user_interface.UserInterfaceBa method*), 494
prompt_question() (*mafw.ui.console_user_interface.ConsoleInterface method*), 496
prompt_question() (*mafw.ui.rich_user_interface.RichInterface method*), 498
prune (*in module mafw.devtools.cli.documentation*), 197
prune_old_versions() (*in module mafw.devtools.documentation*), 229
prune_old_versions() (*in module mafw.devtools.documentation.versions*), 244
push_changes() (*in module mafw.devtools.git*), 248
push_changes() (*in module mafw.devtools.release*), 267
pylock_version_tag() (*in module mafw.devtools.gitlab*), 252
pylock_version_tag() (*in module mafw.devtools.gitlab.registry*), 261
PYPROJECT_FILE (*in module mafw.devtools.dependencies.compile*),

215
PYPROJECT_FILE (*in module mafw.devtools.git*), 248
PyprojectModifier (*class in mafw.devtools.toolchain.pyproject_modifier*), 295
PytestTool (*class in mafw.devtools.toolchain.tools*), 302
PytestTool (*class in mafw.devtools.toolchain.tools.pytest*), 313
python_value() (*mafw.db.fields.FileNameField method*), 179
python_value() (*mafw.db.fields.FileNameListField method*), 179
python_versions_between() (*in module mafw.devtools.dependencies*), 204
python_versions_between() (*in module mafw.devtools.dependencies.compile*), 215
PYTHON_VERSIONS_REQUIREMENTS_FILENAME (*in module mafw.devtools.documentation.requirements*), 240
PyUpgradeTool (*class in mafw.devtools.toolchain.tools.pyupgrade*), 314

Q

queue_batch_size (*mafw.processor.base.Processor attribute*), 378
queue_batch_size (*mafw.processor.Processor attribute*), 365
queue_size (*mafw.processor.base.Processor attribute*), 379
queue_size (*mafw.processor.Processor attribute*), 365
Quit (*mafw.enumerators.LoopingStatus attribute*), 323

R

R (*class in mafw.lazy_import*), 339
R (*class in mafw.steering_gui.controllers.steering_controller*), 437
RangeEditor (*class in mafw.steering_gui.views.filter_editors*), 467
read() (*mafw.devtools.toolchain.precommit_modifier.PreCommitModifier method*), 293
read() (*mafw.devtools.toolchain.pyproject_modifier.PyprojectModifier method*), 297
read_compiled_dependency_versions() (*in module mafw.devtools.dependencies*), 205
read_compiled_dependency_versions() (*in module mafw.devtools.dependencies.compile*), 215
read_current_version() (*in module mafw.devtools.release*), 267
read_current_version() (*in module mafw.devtools.release.versioning*), 276
read_server_metadata() (*in module mafw.devtools.documentation*), 230
read_server_metadata() (*in module mafw.devtools.documentation.server*), 242
recorded_mutation() (*in module mafw.steering_gui.controllers.steering_controller*), 442
redo() (*mafw.steering_gui.controllers.steering_controller.SteeringContr method*), 439
refresh() (*mafw.steering_gui.views.filter_manager.FilterManager method*), 469
refresh() (*mafw.steering_gui.views.group_editor.GroupEditor method*), 470
refresh() (*mafw.steering_gui.views.GroupEditor method*), 459
refresh() (*mafw.steering_gui.views.pipeline_editor.PipelineEditor method*), 471
refresh() (*mafw.steering_gui.views.PipelineEditor method*), 460
refresh() (*mafw.steering_gui.views.steering_tree.SteeringTreeView method*), 474
refresh() (*mafw.steering_gui.views.SteeringTreeView method*), 460
regenerate_versions_json_after_pruning() (*in module mafw.devtools.documentation*), 230
regenerate_versions_json_after_pruning() (*in module mafw.devtools.documentation.versions*), 245
register() (*mafw.devtools.toolchain.registry.ToolRegistry method*), 299
register() (*mafw.devtools.toolchain.ToolRegistry method*), 283
register_db_model_modules() (*in module mafw.hookspecs*), 337
register_db_model_modules() (*in module mafw.plugins*), 351
register_model() (*mafw.db.model_register.ModelRegister method*), 181
register_prefix() (*mafw.db.model_register.ModelRegister method*), 181
register_processors() (*in module mafw.hookspecs*), 337
register_processors() (*in module mafw.plugins*), 351
register_suffix() (*mafw.db.model_register.ModelRegister method*), 182
register_user_interfaces() (*in module mafw.hookspecs*), 337
register_user_interfaces() (*in module mafw.plugins*), 351
RegisteredMeta (*class in mafw.db.db_model*), 174
registry (*in module mafw.devtools.cli.dependencies*), 197
registry (*in module mafw.devtools.cli.documentation*), 198
RELEASE_SECTION_HEADERS (*in module mafw.devtools.release.changelog*), 270
RELEASE_TEMPLATE_FILE (*in module*

mafw.devtools.release.notes), 272
RelPlot (*class in mafw.processor_library.sns_plotter*), 409
remove_filter() (*mafw.steering.builder.SteeringBuilder method*), 427
remove_filter() (*mafw.steering_gui.controllers.steering_controller.SteeringController method*), 440
remove_group() (*mafw.steering.builder.SteeringBuilder method*), 427
remove_group() (*mafw.steering_gui.controllers.steering_controller.SteeringController method*), 440
remove_item() (*mafw.steering_gui.views.filter_tree.FilterTreeView method*), 469
remove_orphan_files (*mafw.processor.base.Processor attribute*), 379
remove_orphan_files (*mafw.processor.Processor attribute*), 365
remove_parameter() (*mafw.steering.builder.SteeringBuilder method*), 427
remove_pipeline_entries() (*mafw.steering_gui.controllers.steering_controller.SteeringController method*), 440
remove_pipeline_entry() (*mafw.steering_gui.controllers.steering_controller.SteeringController method*), 440
remove_processor() (*mafw.steering.builder.SteeringBuilder method*), 427
remove_processor_from_run_list() (*mafw.steering_gui.controllers.steering_controller.SteeringController method*), 440
remove_processor_parameter() (*mafw.steering_gui.controllers.steering_controller.SteeringController method*), 440
remove_widow_db_rows() (*in module mafw.tools.file_tools*), 482
rename_group() (*mafw.steering_gui.controllers.steering_controller.SteeringController method*), 440
rename_processor_entry() (*mafw.steering_gui.controllers.steering_controller.SteeringController method*), 440
render_comparison_json() (*in module mafw.devtools.dependencies*), 205
render_comparison_json() (*in module mafw.devtools.dependencies.compare*), 210
render_comparison_markdown() (*in module mafw.devtools.dependencies*), 205
render_comparison_markdown() (*in module mafw.devtools.dependencies.compare*), 210
render_comparison_rich() (*in module mafw.devtools.dependencies*), 206
render_comparison_rich() (*in module mafw.devtools.dependencies.compare*), 211
render_release_note_section() (*in module mafw.devtools.release*), 268
render_release_note_section() (*in module mafw.devtools.release.notes*), 272
replace_builder() (*mafw.steering_gui.controllers.steering_controller.SteeringController method*), 440
replica_id (*mafw.processor.base.Processor attribute*), 379
replica_id (*mafw.processor.Processor attribute*), 365
replica_name (*mafw.processor.base.Processor property*), 379
replica_name (*mafw.processor.Processor property*), 365
report_build_status() (*in module mafw.devtools.documentation*), 230
report_build_status() (*in module mafw.devtools.documentation.builder*), 238
required_columns (*mafw.processor_library.sns_plotter.SQLPdDataReader attribute*), 413
REQUIREMENTS_GROUPS (*in module mafw.devtools.documentation.requirements*), 240
reset() (*mafw.processor.ActiveParameter method*), 353
reset() (*mafw.processor.parameters.ActiveParameter method*), 382
reset() (*mafw.processor_library.importer.FilenameElement method*), 401
reset() (*mafw.processor_library.importer.FilenameParser method*), 402
reset() (*mafw.steering_gui.controllers.steering_controller.SteeringController method*), 440
reset_processor_parameters() (*mafw.steering_gui.controllers.steering_controller.SteeringController method*), 440
resolve_command() (*mafw.tools.click_extensions.AbbreviateGroup method*), 477
resolve_completion_shell() (*in module mafw.tools.click_extensions*), 480
resolve_config_env() (*in module mafw.tools.toml_tools*), 492
resolve_docs_package_ids_by_version() (*in module mafw.devtools.documentation*), 230
resolve_mafw_docs_package_ids_by_version() (*in module mafw.devtools.gitlab.docs_registry*), 256
resolve_output_format() (*in module mafw.devtools.dependencies*), 206
resolve_output_format() (*in module mafw.devtools.dependencies.compare*), 211
resolve_package_ids_by_version() (*in module mafw.devtools.gitlab*), 252
resolve_package_ids_by_version() (*in module mafw.devtools.gitlab.registry*), 261
resolve_string() (*in module mafw.tools.toml_tools*), 492

resolve_tools() (*in module mafw.devtools.toolchain.filtering*), 292
ReturnValue (*class in mafw.scripts.mafw_exe*), 425
RichInterface (*class in mafw.ui.rich_user_interface*), 497
root_items() (*mafw.steering_gui.models.processor_pipeline_model.ProcessorPipelineModel method*), 453
root_items() (*mafw.steering_gui.models.ProcessorPipelineModel method*), 449
rreplace() (*in module mafw.timer*), 476
RuffTool (*class in mafw.devtools.toolchain.tools*), 303
RuffTool (*class in mafw.devtools.toolchain.tools.ruff*), 316
Run (*mafw.enumerators.ProcessorStatus attribute*), 323
run() (*in module mafw.devtools.documentation*), 230
run() (*in module mafw.devtools.documentation.builder*), 238
run() (*in module mafw.tools.shell_tools*), 488
run() (*mafw.runner.MAFwApplication method*), 416
run() (*mafw.steering_gui.threading.Worker method*), 455
run() (*mafw.steering_gui.threading.workers.Worker method*), 455
run_pip_audit() (*in module mafw.devtools.dependencies*), 206
run_pip_audit() (*in module mafw.devtools.dependencies.audit*), 208
run_processor_list_with_loop_modifier() (*in module mafw.examples.processor_list*), 335
run_simple_processor_list() (*in module mafw.examples.processor_list*), 335
run_stdout() (*in module mafw.tools.shell_tools*), 488
RunnerNotInitialized, 343

S

save() (*mafw.processor_library.abstract_plotter.GenericPlotter method*), 392
save() (*mafw.steering_gui.controllers.steering_controller.SteeringController method*), 440
save_as() (*mafw.steering_gui.controllers.steering_controller.SteeringController method*), 440
search() (*mafw.processor_library.importer.FilenameElement method*), 401
section_key_from_index() (*mafw.steering_gui.models.steering_tree_model.SteeringTreeModel method*), 454
section_key_from_index() (*mafw.steering_gui.models.SteeringTreeModel method*), 450
section_path (*mafw.devtools.toolchain.base.ProjectTool property*), 289
section_path (*mafw.devtools.toolchain.ProjectTool property*), 281
section_path (*mafw.devtools.toolchain.tools.git_cliff.GitCliffTool property*), 308
section_path (*mafw.devtools.toolchain.tools.GitCliffTool property*), 300
section_path (*mafw.devtools.toolchain.tools.mypy.MypyTool property*), 310
section_path (*mafw.devtools.toolchain.tools.MypyTool property*), 301
section_path (*mafw.devtools.toolchain.tools.pip_audit.PipAuditTool property*), 311
section_path (*mafw.devtools.toolchain.tools.PipAuditTool property*), 301
section_path (*mafw.devtools.toolchain.tools.precommit.PreCommitTool property*), 312
section_path (*mafw.devtools.toolchain.tools.PreCommitTool property*), 302
section_path (*mafw.devtools.toolchain.tools.pytest.PytestTool property*), 313
section_path (*mafw.devtools.toolchain.tools.PytestTool property*), 303
section_path (*mafw.devtools.toolchain.tools.pyupgrade.PyUpgradeTool property*), 316
section_path (*mafw.devtools.toolchain.tools.ruff.RuffTool property*), 318
section_path (*mafw.devtools.toolchain.tools.RuffTool property*), 305
section_path (*mafw.devtools.toolchain.tools.sphinx.SphinxTool property*), 320
section_path (*mafw.devtools.toolchain.tools.SphinxTool property*), 306
select_pipeline_item() (*mafw.steering_gui.views.steering_tree.SteeringTreeView method*), 474
select_pipeline_item() (*mafw.steering_gui.views.SteeringTreeView method*), 460
selected_field() (*mafw.steering_gui.dialogs.add_field_filter_dialog.A method*), 444
selected_model() (*mafw.steering_gui.dialogs.add_model_dialog.AddM method*), 444
selected_names() (*mafw.steering_gui.dialogs.add_processor_dialog.A method*), 444
selected_names() (*mafw.steering_gui.dialogs.AddProcessorDialog method*), 442
serialize() (*in module mafw.steering.serializer*), 434
server (*in module mafw.devtools.cli.documentation*), 198
SERVER_DEFAULT_DIRECTORY (*in module mafw.devtools.documentation.server*), 243
server_default_directory() (*in module mafw.devtools.documentation*), 231
server_default_directory() (*in module mafw.devtools.documentation.server*), 242
SERVER_METADATA_FILENAME (*in module mafw.devtools.documentation.server*), 243
server_metadata_path() (*in module mafw.devtools.documentation*), 231
server_metadata_path() (*in module*

mafw.devtools.documentation.server), 242
server_start_impl() (*in module mafw.devtools.documentation.server*), 243
server_stop_by_metadata() (*in module mafw.devtools.documentation.server*), 243
ServerStatusEnum (*class in mafw.devtools.documentation*), 220
ServerStatusEnum (*class in mafw.devtools.documentation.server*), 241
set_analysis_description() (*mafw.steering.builder.SteeringBuilder method*), 427
set_analysis_description() (*mafw.steering_gui.controllers.steering_controller.SteeringController method*), 440
set_analysis_name() (*mafw.steering.builder.SteeringBuilder method*), 428
set_analysis_name() (*mafw.steering_gui.controllers.steering_controller.SteeringController method*), 440
set_available_interfaces() (*mafw.steering_gui.views.ui_editor.UIEditor method*), 474
set_available_interfaces() (*mafw.steering_gui.views.UIEditor method*), 460
set_backend() (*mafw.steering_gui.views.database_editor.AuthenticationWidget method*), 462
set_backend() (*mafw.steering_gui.views.database_editor.ParametersWidget method*), 463
set_backend() (*mafw.steering_gui.views.database_editor.TriggersWidget method*), 463
set_condition() (*mafw.steering_gui.views.filter_editors.ConditionEditor method*), 465
set_condition() (*mafw.steering_gui.views.filter_editors.ConditionEditor method*), 465
set_condition_warning() (*mafw.steering_gui.models.filter_tree_model.FilterTreeModel method*), 452
set_config() (*mafw.steering_gui.views.filter_editors.FieldFilterEditor method*), 465
set_config() (*mafw.steering_gui.views.filter_editors.LogicEditor method*), 466
set_config() (*mafw.steering_gui.views.filter_editors.ModelFieldEditor method*), 467
set_config() (*mafw.steering_gui.views.filter_editors.ProcessorFilterEditor method*), 467
set_create_standard_tables() (*mafw.steering.builder.SteeringBuilder method*), 428
set_create_standard_tables() (*mafw.steering_gui.controllers.steering_controller.SteeringController method*), 440
set_current_interface() (*mafw.steering_gui.views.ui_editor.UIEditor method*), 474
set_current_interface() (*mafw.steering_gui.views.UIEditor method*), 460
set_data() (*mafw.steering_gui.views.database_editor.AuthenticationWidget method*), 462
set_data() (*mafw.steering_gui.views.database_editor.DatabaseEditor method*), 463
set_data() (*mafw.steering_gui.views.database_editor.ParametersWidget method*), 463
set_data() (*mafw.steering_gui.views.DatabaseEditor method*), 459
set_data() (*mafw.steering_gui.views.globals_editor.GlobalsEditor method*), 470
set_data() (*mafw.steering_gui.views.GlobalsEditor method*), 459
set_database() (*mafw.db.trigger.Trigger method*), 189
set_db_attribute() (*mafw.steering.builder.SteeringBuilder method*), 428
set_db_attribute() (*mafw.steering_gui.controllers.steering_controller.SteeringController method*), 440
set_db_authentication() (*mafw.steering.builder.SteeringBuilder method*), 428
set_db_authentication() (*mafw.steering_gui.controllers.steering_controller.SteeringController method*), 440
set_db_parameters() (*mafw.steering.builder.SteeringBuilder method*), 428
set_db_parameters() (*mafw.steering_gui.controllers.steering_controller.SteeringController method*), 440
set_db_pragmas() (*mafw.steering.builder.SteeringBuilder method*), 428
set_db_pragmas() (*mafw.steering_gui.controllers.steering_controller.SteeringController method*), 440
set_db_tables() (*mafw.steering.builder.SteeringBuilder method*), 428
set_db_tables() (*mafw.steering_gui.controllers.steering_controller.SteeringController method*), 440
set_default() (*mafw.steering.builder.SteeringBuilder method*), 428
set_excluded_names() (*mafw.steering_gui.views.filter_editors.FieldNameWidget method*), 465
set_field_type() (*mafw.steering_gui.views.filter_editors.ListEditor method*), 466
set_field_type() (*mafw.steering_gui.views.filter_editors.RangeEditor method*), 467
set_field_type() (*mafw.steering_gui.views.filter_editors.SingleValueEditor method*), 468
set_field_type() (*mafw.steering_gui.views.filter_editors.ValueEditor*

method), 468
set_filter_conditionals() (*mafw.steering.builder.SteeringBuilder method*), 428
set_filter_conditionals() (*mafw.steering_gui.controllers.steering_controller.SteeringController method*), 441
set_filter_config() (*mafw.steering.builder.SteeringBuilder method*), 428
set_filter_config() (*mafw.steering_gui.controllers.steering_controller.SteeringController method*), 441
set_filter_field() (*mafw.steering.builder.SteeringBuilder method*), 428
set_filter_field() (*mafw.steering_gui.controllers.steering_controller.SteeringController method*), 441
set_filter_logic() (*mafw.steering.builder.SteeringBuilder method*), 428
set_filter_logic() (*mafw.steering_gui.controllers.steering_controller.SteeringController method*), 441
set_group_attributes() (*mafw.steering_gui.controllers.steering_controller.SteeringController method*), 441
set_group_description() (*mafw.steering_gui.controllers.steering_controller.SteeringController method*), 441
set_group_item() (*mafw.steering_gui.views.group_editor.GroupEditor method*), 470
set_group_item() (*mafw.steering_gui.views.GroupEditor method*), 459
set_group_processors() (*mafw.steering_gui.controllers.steering_controller.SteeringController method*), 441
set_logic_warning() (*mafw.steering_gui.models.filter_tree_model.FilterTreeModel method*), 452
set_model() (*mafw.steering_gui.views.filter_tree.FilterTreeView method*), 469
set_model() (*mafw.steering_gui.views.group_editor.GroupEditor method*), 470
set_model() (*mafw.steering_gui.views.GroupEditor method*), 459
set_model() (*mafw.steering_gui.views.pipeline_editor.PipelineEditor method*), 472
set_model() (*mafw.steering_gui.views.PipelineEditor method*), 460
set_model() (*mafw.steering_gui.views.steering_tree.SteeringTreeView method*), 474
set_model() (*mafw.steering_gui.views.SteeringTreeView method*), 460
set_model_info() (*mafw.steering_gui.views.filter_editors.FieldNameWidget method*), 465
set_name() (*mafw.steering_gui.views.filter_editors.FieldNameWidget method*), 465
set_name() (*mafw.steering_gui.views.filter_editors.FreeNameWidget method*), 466
set_new_only() (*mafw.steering.builder.SteeringBuilder method*), 428
set_new_only() (*mafw.steering_gui.controllers.steering_controller.SteeringController method*), 441
set_parameter() (*mafw.steering.builder.SteeringBuilder method*), 428
set_parameter_value() (*mafw.processor.base.Processor method*), 370
set_parameter_value() (*mafw.processor.Processor method*), 362
set_parameters() (*mafw.steering_gui.views.processor_parameter_editor.ProcessorParameterEditor method*), 473
set_pipeline() (*mafw.steering_gui.models.processor_pipeline_model.ProcessorPipelineModel method*), 453
set_pipeline() (*mafw.steering_gui.models.ProcessorPipelineModel method*), 449
set_pipeline() (*mafw.steering_gui.models.steering_tree_model.SteeringTreeModel method*), 454
set_pipeline() (*mafw.steering_gui.models.SteeringTreeModel method*), 450
set_processor_data() (*mafw.steering_gui.views.processor_parameter_editor.ProcessorParameterEditor method*), 473
set_processor_data() (*mafw.steering_gui.views.ProcessorParameterEditor method*), 460
set_processor_filters() (*mafw.steering.builder.SteeringBuilder method*), 428
set_processor_filters() (*mafw.steering_gui.controllers.steering_controller.SteeringController method*), 441
set_processor_new_only() (*mafw.steering.builder.SteeringBuilder method*), 428
set_processor_new_only() (*mafw.steering_gui.controllers.steering_controller.SteeringController method*), 441
set_processor_parameter() (*mafw.steering_gui.controllers.steering_controller.SteeringController method*), 441
set_processors_to_run() (*mafw.steering.builder.SteeringBuilder method*), 428
set_processors_to_run() (*mafw.steering_gui.controllers.steering_controller.SteeringController method*), 441
set_replica_inheritance() (*mafw.steering.builder.SteeringBuilder method*), 428
set_replica_inheritance() (*mafw.steering_gui.controllers.steering_controller.SteeringController method*), 441
set_user_interface()

(*mafw.steering.builder.SteeringBuilder method*), 428
set_ui_interface() (*mafw.steering_gui.controllers.steering_controller.SteeringController method*), 441
set_url() (*mafw.steering_gui.views.database_editor.TransportWidget method*), 463
set_validation_level() (*mafw.steering_gui.controllers.steering_controller.SteeringController method*), 441
set_validation_level() (*mafw.steering_gui.views.meta_editor.MetadataEditor method*), 471
set_validation_level() (*mafw.steering_gui.views.MetadataEditor method*), 459
set_value() (*mafw.steering_gui.views.filter_editors.ListEditor method*), 466
set_value() (*mafw.steering_gui.views.filter_editors.RangeEditor method*), 467
set_value() (*mafw.steering_gui.views.filter_editors.SingleValueEditor method*), 468
set_value() (*mafw.steering_gui.views.filter_editors.ValueEditorBase method*), 468
set_value() (*mafw.steering_gui.views.processor_parameter_editor.MyPdEditor method*), 473
shortcut (*mafw.steering_gui.actions.ActionSpec attribute*), 435
show_exception_dialog() (*in module mafw.steering_gui.utils.exception_handling*), 457
single_loop() (*in module mafw.decorators*), 194
SingleLoop (*mafw.enumerators.LoopType attribute*), 322
singleton() (*in module mafw.decorators*), 194
SingleValueEditor (*class in mafw.steering_gui.views.filter_editors*), 467
Skip (*mafw.enumerators.LoopingStatus attribute*), 323
skip_item() (*mafw.examples.loop_modifier.ModifyLoopProcessor method*), 335
skip_item() (*mafw.processor.base.Processor method*), 376
skip_item() (*mafw.processor.Processor method*), 362
skipped_items (*mafw.examples.loop_modifier.ModifyLoopProcessor attribute*), 335
slice_data_frame() (*in module mafw.tools.pandas_tools*), 485
slice_data_frame() (*mafw.processor_library.sns_plotter.SNSPlotter method*), 412
slicing_dict (*mafw.processor_library.sns_plotter.SNSPlotter attribute*), 413
SNSFigurePlotter (*class in mafw.processor_library.sns_plotter*), 410
SNSPlotter (*class in mafw.processor_library.sns_plotter*), 410
soft_recreate (*mafw.processor_library.db_init.TableCreator attribute*), 396
sort_tags_semver() (*in module mafw.devtools.documentation*), 231
sort_tags_semver() (*in module mafw.devtools.git*), 248
SPHINX_BUILD_CMD (*in module mafw.devtools.documentation.builder*), 239
SphinxTool (*class in mafw.devtools.toolchain.tools*), 305
SphinxTool (*class in mafw.devtools.toolchain.tools.sphinx*), 319
sql_files (*mafw.processor_library.db_init.SQLScriptRunner attribute*), 394
sqlite_memory_button (*mafw.steering_gui.views.database_editor.TransportWidget property*), 463
sqlite_open_button (*mafw.steering_gui.views.database_editor.TransportWidget property*), 463
SQLiteSelect (*class in mafw.db.trigger*), 186
SQLPdDataRetriever (*class in mafw.processor_library.sns_plotter*), 413
SQLScriptRunner (*class in mafw.processor_library.db_init*), 393
STABLE_TAG_PATTERN (*in module mafw.devtools.git*), 248
STABLE_TAG_PATTERN (*in module mafw.devtools.release.versioning*), 277
StandardTable (*class in mafw.db.std_tables*), 183
StandardTableDoesNotExist, 182
Start (*mafw.enumerators.ProcessorStatus attribute*), 323
start() (*mafw.examples.db_processors.CountStandardTables method*), 325
start() (*mafw.examples.db_processors.FillFileTableProcessor method*), 326
start() (*mafw.examples.importer_example.ImporterExample method*), 328
start() (*mafw.examples.loop_modifier.FindNPrimeNumber method*), 330
start() (*mafw.examples.loop_modifier.FindPrimeNumberInRange method*), 331
start() (*mafw.examples.loop_modifier.FindPrimeNumberInRangeParal method*), 332
start() (*mafw.examples.loop_modifier.FindPrimeNumberInRangeParal method*), 333
start() (*mafw.examples.loop_modifier.FindPrimeNumberInRangeTunab method*), 334
start() (*mafw.examples.loop_modifier.ModifyLoopProcessor method*), 335
start() (*mafw.examples.sum_processor.AccumulatorProcessor method*), 336
start() (*mafw.examples.sum_processor.GaussAdder method*), 337
start() (*mafw.processor.base.Processor method*),

376
start() (*mafw.processor.Processor method*), 362
start() (*mafw.processor_library.db_init.SQLScriptRunner method*), 394
start() (*mafw.processor_library.db_init.TableCreator method*), 396
start() (*mafw.processor_library.db_init.TriggerRefresher method*), 399
start() (*mafw.processor_library.importer.Importer method*), 404
start() (*mafw.processor_library.sns_plotter.SNSPlotter method*), 412
status_tip (*mafw.steering_gui.actions.ActionSpec attribute*), 435
std_upsert() (*mafw.db.db_model.MAFwBaseModel class method*), 172
std_upsert_many() (*mafw.db.db_model.MAFwBaseModel class method*), 173
STEERING_FILE
    mafw-run command line option, 423
    mafw-steering command line option, 424
SteeringBuilder (*class in mafw.steering.builder*), 427
SteeringController (*class in mafw.steering_gui.controllers.steering_controller*), 437
SteeringControllerError, 436
SteeringFileDeprecation, 343
SteeringTextEditor (*class in mafw.steering_gui.dialogs*), 442
SteeringTextEditor (*class in mafw.steering_gui.dialogs.steering_text_editor*), 445
SteeringTreeModel (*class in mafw.steering_gui.models*), 449
SteeringTreeModel (*class in mafw.steering_gui.models.steering_tree_model*), 454
SteeringTreeRoles (*class in mafw.steering_gui.models*), 450
SteeringTreeRoles (*class in mafw.steering_gui.models.steering_tree_model*), 454
SteeringTreeView (*class in mafw.steering_gui.views*), 460
SteeringTreeView (*class in mafw.steering_gui.views.steering_tree*), 474
Successful (*mafw.enumerators.ProcessorExitStatus attribute*), 323
summarize_freeze_changes() (*in module mafw.devtools.dependencies.freeze*), 218
supports_safe_create() (*mafw.db.trigger.MySQLDialect method*), 185
supports_safe_create() (*mafw.db.trigger.PostgreSQLDialect method*), 186
supports_safe_create() (*mafw.db.trigger.SQLiteDialect method*), 186
supports_safe_create() (*mafw.db.trigger.TriggerDialect method*), 189
supports_trigger_type() (*mafw.db.trigger.MySQLDialect method*), 185
supports_trigger_type() (*mafw.db.trigger.PostgreSQLDialect method*), 186
supports_trigger_type() (*mafw.db.trigger.SQLiteDialect method*), 186
supports_trigger_type() (*mafw.db.trigger.TriggerDialect method*), 189
supports_update_of_columns() (*mafw.db.trigger.MySQLDialect method*), 185
supports_update_of_columns() (*mafw.db.trigger.PostgreSQLDialect method*), 186
supports_update_of_columns() (*mafw.db.trigger.SQLiteDialect method*), 186
supports_update_of_columns() (*mafw.db.trigger.TriggerDialect method*), 190
supports_when_clause() (*mafw.db.trigger.MySQLDialect method*), 185
supports_when_clause() (*mafw.db.trigger.PostgreSQLDialect method*), 186
supports_when_clause() (*mafw.db.trigger.SQLiteDialect method*), 186
supports_when_clause() (*mafw.db.trigger.TriggerDialect method*), 190
suppress_warnings() (*in module mafw.decorators*), 194

T

T (*class in mafw.lazy_import*), 339
T (*class in mafw.steering_gui.threading.workers*), 455
table (*mafw.steering_gui.views.database_editor.ParametersWidget property*), 463
table_name (*mafw.processor_library.sns_plotter.SQLPdDataRetriever attribute*), 413
TableCreator (*class in mafw.processor_library.db_init*), 394
text (*mafw.steering_gui.actions.ActionSpec attribute*), 435
ThreadExceptionHandler (*class in mafw.steering_gui.utils.exception_handling*),

457
Timer (*class in mafw.timer*), 475
to_config_dict() (*mafw.steering.builder.SteeringBuilder method*), 428
to_dict() (*mafw.db.db_model.MAFwBaseModel method*), 173
to_document() (*mafw.steering.builder.SteeringBuilder method*), 429
to_expression() (*mafw.db.db_filter.ConditionalFilterCondition method*), 159
to_expression() (*mafw.db.db_filter.ConditionalNode method*), 159
to_expression() (*mafw.db.db_filter.ConditionNode method*), 157
to_expression() (*mafw.db.db_filter.LogicalNode method*), 162
to_schema() (*mafw.processor.ActiveParameter method*), 353
to_schema() (*mafw.processor.parameters.ActiveParameter method*), 382
Token (*in module mafw.db.db_filter*), 169
TOKEN_SPECIFICATION (*in module mafw.db.db_filter*), 169
tokenize() (*in module mafw.db.db_filter*), 168
TomlSyntaxHighlighter (*class in mafw.steering_gui.dialogs.steering_text_editor*), 445
ToolChainTool (*class in mafw.devtools.toolchain*), 281
ToolChainTool (*class in mafw.devtools.toolchain.base*), 289
ToolCheckResult (*class in mafw.devtools.toolchain*), 282
ToolCheckResult (*class in mafw.devtools.toolchain.base*), 290
ToolRegistry (*class in mafw.devtools.toolchain*), 283
ToolRegistry (*class in mafw.devtools.toolchain.registry*), 298
ToolUpdateResult (*class in mafw.devtools.toolchain*), 284
ToolUpdateResult (*class in mafw.devtools.toolchain.base*), 291
TransportWidget (*class in mafw.steering_gui.views.database_editor*), 463
Trigger (*class in mafw.db.trigger*), 186
TriggerAction (*class in mafw.db.trigger*), 189
TriggerDialect (*class in mafw.db.trigger*), 189
TriggerDisabler (*class in mafw.db.std_tables*), 183
triggered (*mafw.steering_gui.actions.ActionSpec attribute*), 435
TriggerRefresher (*class in mafw.processor_library.db_init*), 396
triggers() (*mafw.db.db_model.MAFwBaseModel class method*), 173
TriggerStatus (*class in mafw.db.std_tables*), 184
TriggerStatusDoesNotExist, 183
TriggerWhen (*class in mafw.db.trigger*), 190
tuning_loop_type (*mafw.tools.loop_tuning.LoopTuningMixin attribute*), 484
tuning_max_workers (*mafw.tools.loop_tuning.LoopTuningMixin attribute*), 484
tuning_queue_batch_size (*mafw.tools.loop_tuning.LoopTuningMixin attribute*), 484
tuning_queue_size (*mafw.tools.loop_tuning.LoopTuningMixin attribute*), 485
type_lut (*mafw.processor_library.importer.FilenameElement attribute*), 401

U

ui_dict (*mafw.plugin_manager.LoadedPlugins attribute*), 348
ui_list (*mafw.plugin_manager.LoadedPlugins attribute*), 349
UIConfiguration (*class in mafw.steering.models*), 433
UIEditor (*class in mafw.steering_gui.views*), 460
UIEditor (*class in mafw.steering_gui.views.ui_editor*), 474
undo() (*mafw.steering_gui.controllers.steering_controller.SteeringContr method*), 441
UnexpectedEndOfExpressionError, 157
UnexpectedTokenError, 157
unfreeze_dependencies() (*in module mafw.devtools.dependencies*), 207
unfreeze_dependencies() (*in module mafw.devtools.dependencies.freeze*), 218
unfreeze_pyproject_toml() (*in module mafw.devtools.dependencies*), 207
unfreeze_pyproject_toml() (*in module mafw.devtools.dependencies.freeze*), 218
unfreeze_requirement() (*in module mafw.devtools.dependencies*), 207
unfreeze_requirement() (*in module mafw.devtools.dependencies.freeze*), 219
uninstall_completion_files() (*in module mafw.tools.click_extensions*), 480
unique_id (*mafw.processor.base.Processor attribute*), 379
unique_id (*mafw.processor.Processor attribute*), 365
unique_name (*mafw.processor.base.Processor property*), 379
unique_name (*mafw.processor.Processor property*), 365
Unknown (*mafw.enumerators.ProcessorStatus attribute*), 323
UnknownDBEngine, 343
UnknownField (*class in mafw.db.db_wizard*), 176
UnknownGroupMemberIssue, 343
UnknownNameError, 157
UnknownProcessor, 343
UnknownProcessorGroup, 343
UnknownProcessorsToRunEntryIssue, 343
UnsupportedDatabaseError, 343

unwrap() (*mafw.tools.toml_tools.PathItem method*), 489
update() (*mafw.devtools.toolchain.base.HostTool method*), 286
update() (*mafw.devtools.toolchain.base.ProjectTool method*), 288
update() (*mafw.devtools.toolchain.base.ToolChainTool method*), 290
update() (*mafw.devtools.toolchain.HostTool method*), 278
update() (*mafw.devtools.toolchain.ProjectTool method*), 281
update() (*mafw.devtools.toolchain.ToolChainTool method*), 282
update() (*mafw.devtools.toolchain.tools.pyupgrade.PyUpgradeTool method*), 315
update() (*mafw.devtools.toolchain.tools.ruff.RuffTool method*), 317
update() (*mafw.devtools.toolchain.tools.RuffTool method*), 304
update() (*mafw.devtools.toolchain.tools.uv.UvTool method*), 320
update() (*mafw.devtools.toolchain.tools.UvTool method*), 306
update_context() (*mafw.steering_gui.views.filter_editors.FreeNameWidget method*), 466
update_db() (*mafw.processor_library.abstract_plotter.GenericPlotter method*), 392
update_dependency_list() (*in module mafw.devtools.dependencies.freeze*), 219
update_doc_target_version() (*in module mafw.devtools.release*), 268
update_doc_target_version() (*in module mafw.devtools.release.versioning*), 276
update_from_dict() (*mafw.db.db_model.MAFwBaseModel method*), 174
update_lower_bound() (*mafw.devtools.toolchain.pyproject_modifier.PyprojectModifier method*), 297
update_notice_version() (*in module mafw.devtools.release*), 268
update_notice_version() (*in module mafw.devtools.release.versioning*), 276
update_requirements_and_readme() (*in module mafw.devtools.dependencies*), 207
update_requirements_and_readme() (*in module mafw.devtools.dependencies.freeze*), 219
update_rev() (*mafw.devtools.toolchain.precommit_modifier.PreCommitModifier method*), 293
update_summary() (*mafw.steering_gui.views.meta_editor.MetadataEditor method*), 471
update_summary() (*mafw.steering_gui.views.MetadataEditor method*), 459
update_task() (*mafw.ui.abstract_user_interface.UserInterfaceBase method*), 495
update_task() (*mafw.ui.console_user_interface.ConsoleInterface method*), 496
update_task() (*mafw.ui.rich_user_interface.RichInterface method*), 499
upload_docs_zip_to_gitlab_generic_registry() (*in module mafw.devtools.documentation*), 231
upload_docs_zip_to_gitlab_generic_registry() (*in module mafw.devtools.gitlab.docs_registry*), 256
upload_generic_file() (*in module mafw.devtools.gitlab*), 252
upload_generic_file() (*in module mafw.devtools.gitlab.registry*), 262
url_edit (*mafw.steering_gui.views.database_editor.TransportWidget property*), 463
UserInterfaceBase (*class in mafw.ui.abstract_user_interface*), 493
UserInterfaceClassProtocol (*class in mafw.lazy_import*), 340
UserInterfaceMeta (*class in mafw.ui.abstract_user_interface*), 495
username_edit (*mafw.steering_gui.views.database_editor.AuthenticatioProperty*), 462
UvTool (*class in mafw.devtools.toolchain.tools*), 306
UvTool (*class in mafw.devtools.toolchain.tools.uv*), 320

V

VALID_AUTH_METHODS (*in module mafw.db.db_connection*), 155
VALID_HATCH_SEGMENTS (*in module mafw.devtools.release.versioning*), 277
validate() (*in module mafw.steering.validation*), 434
validate() (*mafw.steering.builder.SteeringBuilder method*), 429
validate() (*mafw.steering_gui.controllers.steering_controller.SteeringC method*), 441
validate_configuration() (*mafw.processor.base.Processor method*), 376
validate_configuration() (*mafw.processor.Processor method*), 363
validate_configuration() (*mafw.processor_library.db_init.SQLScriptRunner method*), 394
validate_configuration() (*mafw.processor_library.db_init.TableCreator method*), 396
validate_database_conf() (*in module mafw.processor*), 367
validate_database_conf() (*in module mafw.processor.utils*), 387
validate_doc_target_version() (*in module mafw.devtools.release.versioning*), 276
validate_error_message (*mafw.devtools.release.DocTargetVersionPrompt attribute*), 263
validate_error_message (*mafw.devtools.release.versioning.DocTargetVersionPrompt attribute*), 274

validate_item() (*mafw.processor.processor_list.ProcessorList static method*), 385
validate_item() (*mafw.processor.ProcessorList static method*), 366
validate_items() (*mafw.processor.processor_list.ProcessorList static method*), 386
validate_items() (*mafw.processor.ProcessorList static method*), 366
validation_level() (*mafw.steering_gui.controllers.steering_controller.SteeringGui method*), 441
ValidationIssue, 343
ValidationIssuesDialog (*class in mafw.steering_gui.dialogs.validation_issues_dialog*), 445
ValidationLevel (*class in mafw.steering.builder*), 429
value (*mafw.processor.parameters.PassiveParameter property*), 383
value (*mafw.processor.PassiveParameter property*), 354
value (*mafw.processor_library.importer.FilenameElement property*), 401
value() (*mafw.steering_gui.views.processor_parameter_editor.PathEditor method*), 473
ValueEditorBase (*class in mafw.steering_gui.views.filter_editors*), 468
verify() (*mafw.devtools.toolchain.base.HostTool method*), 286
verify() (*mafw.devtools.toolchain.base.ProjectTool method*), 289
verify() (*mafw.devtools.toolchain.base.ToolChainTool method*), 290
verify() (*mafw.devtools.toolchain.HostTool method*), 278
verify() (*mafw.devtools.toolchain.ProjectTool method*), 281
verify() (*mafw.devtools.toolchain.ToolChainTool method*), 282
verify() (*mafw.devtools.toolchain.tools.precommit.PreCommitTool method*), 312
verify() (*mafw.devtools.toolchain.tools.PreCommitTool method*), 302
verify() (*mafw.devtools.toolchain.tools.pyupgrade.PyUpgradeTool method*), 315
verify() (*mafw.devtools.toolchain.tools.ruff.RuffTool method*), 318
verify() (*mafw.devtools.toolchain.tools.RuffTool method*), 305
verify() (*mafw.devtools.toolchain.tools.uv.UvTool method*), 321
verify() (*mafw.devtools.toolchain.tools.UvTool method*), 307
verify_checksum() (*in module mafw.tools.file_tools*), 483
verify_lowest_resolution() (*in module mafw.devtools.dependencies*), 207
verify_lowest_resolution() (*in module mafw.devtools.dependencies.verify*), 220
VERSION_PATTERN (*in module mafw.devtools.release.versioning*), 277
VersionComparisonResult (*class in mafw.devtools.dependencies*), 200
VersionComparisonResult (*class in mafw.devtools.dependencies.compare*), 209
VersionController (*in module mafw.devtools.release.versioning*), 277

W

where_clause (*mafw.processor_library.sns_plotter.SQLPdDataRetrieve attribute*), 414
while_condition() (*mafw.examples.loop_modifier.FindNPrimeNumbe method*), 330
while_condition() (*mafw.processor.base.Processor method*), 376
while_condition() (*mafw.processor.Processor method*), 363
while_loop() (*in module mafw.decorators*), 194
WhileLoop (*mafw.enumerators.LoopType attribute*), 322
Worker (*class in mafw.steering_gui.threading*), 454
Worker (*class in mafw.steering_gui.threading.workers*), 455
WorkerSignals (*class in mafw.steering_gui.threading.workers*), 455
write() (*mafw.devtools.toolchain.precommit_modifier.PreCommitModif method*), 293
write() (*mafw.devtools.toolchain.pyproject_modifier.PyprojectModifier method*), 297
write() (*mafw.steering.builder.SteeringBuilder method*), 429
write_legacy_redirect_page() (*in module mafw.devtools.documentation*), 232
write_legacy_redirect_page() (*in module mafw.devtools.documentation.versions*), 245
write_redirect_page() (*in module mafw.devtools.documentation*), 232
write_redirect_page() (*in module mafw.devtools.documentation.versions*), 245
write_redirects_file() (*in module mafw.devtools.documentation*), 232
write_redirects_file() (*in module mafw.devtools.documentation.versions*), 245
write_root_landing_page() (*in module mafw.devtools.documentation*), 232
write_root_landing_page() (*in module mafw.devtools.documentation.versions*), 245
write_server_metadata() (*in module mafw.devtools.documentation*), 232
write_server_metadata() (*in module*

mafw.devtools.documentation.server), 243
write_versions_json() (*in module mafw.devtools.documentation*), 232
write_versions_json() (*in module mafw.devtools.documentation.versions*), 245