Scythe Documentation

Release 0.1.1

Materials Data Facility Team, Citrine Informatics

Jan 09, 2023

CONTENTS:

Scythe is a library of tools that generate summaries of the data contained in scientific data files. The goal of Scythe is to provide a shared resources of these tool to avoid duplication of effort between the many emerging scientific databases. Each extractor is designed to generate the sum of all data needed by each of these databases with a uniform API so that specific projects can write simple adaptors for their needs.

Source Code: https://github.com/materials-data-facility/Scythe

CHAPTER

ONE

PROJECT GOALS

The goal of Scythe is to minimize the amount of code duplication between scientific databases. Many databases rely on custom software to extract information from scientific files and transform that data into a standardized format. Automation or analysis software also require extracting information from files. While the data needs of application vary, they all rely on similar algorithms to extract information from the same types of files. *Scythe is designed to be a shared repository for these algorithms*.

The core of Scythe is a collection of "extractors" which each generate simplified, standardized data from a certain class of files. For example, the `ElectronMicroscopyExtractor` produces structured data from file types specific to brands of electron microscopes.

Each extractor does not necessarily generate data in a format needed by any tool. Rather, the extractors are designed to produce *all* of the information needed by all projects that utilize the libraries. In this way, the extractors can service every user without modification.

1.1 What Does Scythe *Do*?

Scythe is designed to provide the answer to two limited questions:

1. ***Which files can I parse with a certain tool?***
   Scythe provides tools for quickly finding files of a certain type
2. ***What information does a set of files contain?***
   Scythe provides a library of tools that transform data into a simpler formats

1.2 What Does Scythe *Not Do*?

There are several questions that are specifically out-of-scope for Scythe:

1. ***How do I get access to files that I want to parse?***
   Scythe does not solve the data transfer problem
2. ***How can I parse large numbers of files reliably?***
   Scythe is not a distributed workflow engine, but is designed to integrate with one for extracting metadata from large filesystems.
3. ***How can I translate data into the schema needed for my application?***
   The goal of Scythe is to go from opaque to well-documented formats. We recommend implementing separate "adapter" classes to transform Scythe metadata to your specific requirements.

See our "how to use Scythe" documentation for more detail on how to integrate Scythe into an application that provides these intentionally-missing features.

CHAPTER

TWO

USER GUIDE

In this part of the guide, we show a simple example of using a Scythe extractor and discuss the full functionality of an extractor.

2.1 Installing Scythe (for users)

Installing Scythe should be as easy as a single `pip` command. Assuming you have a version of Python that is 3.8 or higher, running:

```
pip install scythe-extractors
```

Should get the basics of Scythe installed. By default however, only a small subset of extractors will be installed (this is done so you do not need to install all the dependencies of extractors you may never use). To install additional extractors, you can specify "extras" at install time using the `[...]` syntax for `pip`. For example, if you want to install all the extractors bundled with Scythe (and their dependencies), run:

```
pip install pip install scythe-extractors[all]
```

This will pull in many more packages, but also enable as many extractors as possible. Check the list under `[tool.poetry.extras]` in `pyproject.toml` to see all the options you can specify in the brackets of the `pip install` command.

2.2 Discovering an extractor

Scythe uses stevedore to manage a collection of extractors, and has a utility function for listing available extractors:

```
from scythe.utils.interface import get_available_extractors
print(get_available_extractors())
```

This snippet will print a dictionary of extractors installed on your system. Both extractors that are part of the Scythe base package and those defined by other packages will be included in this list.

2.3 Simple Interface

The methods in *scythe.utils.interface* are useful for most applications. As an example, we illustrate the use of *scythe.file.GenericFileExtractor*, which is available through the `'generic'` extractor plugin:

```
from scythe.utils.interface import execute_extractor
print(execute_extractor('generic', ['pyproject.toml']))
```

The above snippet creates the extractor object and runs it on a file named `pyproject.toml`. Run in the root directory of the Scythe, it would produce output similar to the following, likely with a different `sha512` value if the contents of that file have changed since this documentation was written:

```
[{
    "data_type": "ASCII text",
    "filename": "pyproject.toml",
    "length": 2421,
    "mime_type": "text/plain",
    "path": "pyproject.toml",
    "sha512":
↪"a7eb382c4a3e6cf469656453f9ff2e3c1ac2c02c9c2ba31c3d569a09883e2b2471801c39125dafb7c13bfcaf9cf6afbab92a
↪"
}]
```

The other pre-built parsing function provides the ability to run all extractors on all files in a directory:

```
from scythe.utils.interface import run_all_extractors
gen = run_all_extractors('.')
for record in gen:
    print(record)
```

A third route for using `scythe` is to employ the `get_extractor` operation to access a specific extractor, and then use its class interface (described below):

```
from scythe.utils.interface import get_extractor
extractor = get_extractor('generic')
gen = extractor.parse_directory('.')
for record in gen:
    print(record)
```

2.3.1 Advanced Usage: Adding Context

The function interface for Scythe supports using “context” and “adapters” to provide additional information Scythe into Applications <#id1>`_. Here, we describe the purpose of context and how to use it in our interface.

Context is information about the data held in a file that is not contained within the file itself . Examples include human-friendly descriptions of columns names or which values actually represent a missing measurement in tabular data file (e.g., CSV files). A limited number of extractors support context and this information can be provided via the `execute_extractor` function:

```
execute_extractor('csv', 'tests/data/test.csv', context={'na_values': ['N/A']})
```

The types of context information used by an extractor, if any, is described in the documentation for each extractor.

The `run_all_extractors_on_directory` function has several options for providing context to the extractors. These options include specifying "global context" to be passed to every extractor or adapter and ways of limiting the metadata to specific extractors. See *scythe.utils.interface.run_all_extractors_on_directory()* for further details on the syntax for this command.

Note: *Context is still an experimental feature and APIs are subject to change*

2.4 Class Interface

The class API of extractors provide access to more detailed features of individual extractors. The functionality of an extractor is broken into several simple operations.

2.4.1 Initializing an extractor

The first step to using an extractor is to initialize it. Most extractors do not have any options for the initializer, so you can create them with:

```
extractor = Extractor()
```

Some extractors require configuration options that define how the extractor runs, such as the location of a non-Python executable.

2.4.2 Parsing Method

The main operation for any extractor is the data extraction operation: `parse`.

In most cases, the `parse` operation takes the path to a file and and returns a summary of the data the file holds:

```
metadata = extractor.parse(['/my/file'])
```

Some extractors take multiple files that describe the same object (e.g., the input and output files of a simulation) and use them to generate a single metadata record:

```
metadata = extractor.parse(['/my/file.in', '/my/file.out'])
```

The *grouping method* for these extractors provides logic to identify groups of related files.

Some extractors also can use information that is not contained within the file themselves, which can be provided to the extractor as a "context":

```
metadata = extractor.parse(['/my/file1'], context={'headers': {'temp': 'temperature'}})
```

The documentation for the extractor should indicate valid types of context information.

2.4.3 Grouping Files

Extractors also provide the ability to quickly find groups of associated files: `group`. The `group` operation takes path or list of files and, optionally, directories and generates a list of files that should be treated together when parsing:

```
extractor.group(['input.file', 'output.file', 'unrelated']) # -> [('input.file', 'output.
↪file'), ('unrelated',)]
```

2.4.4 Parsing Entire Directories

`scythe` also provides a utility operation to parse all groups of valid files in a directory:

```
metadata = list(extractor.parse_directory('.'))
```

`parse_directory` is a generator function, so we use `list` here to turn the output into a list format.

2.4.5 Attribution Functions

Two functions, `citations` and `implementors`, are available to determine who contirbuted a extractor. `implementors` returns the list of people who created an extractor, who are likely the points-of-contact for support. `citations` indicates if any publications are available that describe the underlying methods and should be reference in scientific articles.

2.4.6 Full Extractor API

The full API for the extractors are described as a Python abstract class:

class scythe.base.**BaseExtractor**

Abstract base class for a metadata extractor

This class defines the interface for all extractors in Scythe. Each new extractor must implement the `parse()`, *version()*, and *implementors()* functions. The *group()* method should be overridden to generate smart groups of file (e.g., associating the inputs and outputs to the same calculation) *citations()* can be used if there are papers that should be cited if the extractor is used as part of a scientific publication.

See the Scythe Contributor Guide for further details.

identify_files(*path: str*, *context: dict | None = None*) → Iterator[Tuple[str]]

Identify all groups of files likely to be compatible with this extractor

Uses the *group()* function to determine groups of files that should be parsed together.

Parameters

- **path** (*str*) – Root of directory to group together
- **context** (*dict*) – Context about the files

Yields

([str]) Groups of eligible files

extract_directory(*path: str*, *context: dict | None = None*) → Iterator[Tuple[Tuple[str], dict]]

Run extractor on all appropriate files in a directory

Skips files that throw exceptions while parsing

Parameters

- **path** (*str*) – Root of directory to extract metadata from
- **context** (*dict*) – Context about the files

Yields

([str], dict) – Tuple of the group identity and the metadata unit

abstract extract(*group: Iterable[str]*, *context: dict | None = None*) → dict

Extract metadata from a group of files

A group of files is a set of 1 or more files that describe the same object and will be used together to create a single metadata record.

Parameters

- **group** (*[str]*) – A list of one or more files that should be parsed together
- **context** (*dict*) – Context about the files

Returns

The parsed results, in JSON-serializable format.

Return type

(dict)

group(*files: str | List[str]*, *directories: List[str] | None = None*, *context: dict | None = None*) → Iterator[Tuple[str, ...]]

Identify a groups of files and directories that should be parsed together

Will create groups using only the files and directories included as input.

The files of files are _all_ files that could be read by this extractor, which may include many false positives.

Parameters

- **files** (*str or [str]*) – List of files to consider grouping
- **directories** (*[str]*) – Any directories to consider group as well
- **context** (*dict*) – Context about the files

Yields

((str)) – Groups of files

citations() → List[str]

Citation(s) and reference(s) for this extractor

Returns

each element should be a string citation in BibTeX format

Return type

([str])

abstract implementors() → List[str]

List of implementors of the extractor

These people are the points-of-contact for addressing errors or modifying the extractor

Returns

List of implementors in the form "FirstName LastName <email@provider>"

Return type

([str])

abstract version() → str

Return the version of the extractor

Returns
Version of the extractor

Return type
(str)

property schema: dict

Schema for the output of the extractor

2.5 Integrating Scythe into Applications

Scythe is designed to create a documented, JSON-format version of scientific files, but these files might not yet be in a form useful for your application. We recommend an "adapter" approach to post-process these "generic JSON" files that can actually be used for your application.

2.5.1 BaseAdapter

The `BaseAdapter` class defines the interface for all adapters.

class scythe.adapters.base.**BaseAdapter**

Template for tools that transform metadata into a new form

abstract transform(*metadata: dict*, *context: None | dict = None*) → Any

Process metadata into a new form

Parameters

- **metadata** (*dict*) – Metadata to transform
- **context** (*dict*) – Any context information used during transformation

Returns
Metadata in a new form, can be any type of object. `None` corresponding

check_compatibility(*parser:* BaseExtractor) → bool

Evaluate whether an adapter is compatible with a certain parser

Parameters
parser (BaseExtractor) – Parser to evaluate

Returns
(bool) Whether this parser is compatible

version() → None | str

Version of the parser that an adapter was created for

Returns

(str) Version of parser this adapter was designed for,
or `None` if not applicable

Adapters must fulfill a single operation, `transform`, which renders metadata from one of the Scythe extractors into a new form. There are no restrictions on the output for this function, except that `None` indicates that there is no valid transformation for an object.

The `check_compatibility` and `version` method provide a route for marking which versions of an extractor are compatible with an adapter. `scythe` uses the version in utility operations to provide warnings to users about when an adapter is out-of-date.

2.5.2 Using Adapters

The same utility operations *described above* support using adapters. The `execute_extractor` function has an argument, `adapter`, that takes the name of the adapter as an input and causes the parsing operation to run the adapter after parsing. The `run_all_extractors` function also has arguments (e.g., `adapter_map`) that associate each extractor with the adapter needed to run after parsing.

As an example, we will demonstrate an adapter that comes packaged with Scythe: *`scythe.adapters.base.SerializeAdapter`* The serialize adapter is registered using `stevedore` as the name "serialize". To use it after all extractors:

```
from scythe.utils.interface import run_all_extractors
gen = run_all_extractors('.', default_adapter='serialize')
```

2.5.3 Implementing Adapters

Any new adapters must inherit from the `BaseAdapter` class defined above. You only need implement the `transform` operation.

Once the adapter is implemented, you need to put it in a project that is installable via pip. See [python docs](https://docs.python.org/3.7/distutils/setupscript.html) for a detailed tutorial or copy the structure used by the MDF's adapter library.

Then, register the adapter with `stevedore` by adding it as an entry point in your project's `setup.py` or `pyproject.toml` file. See the stevedore documentation for more detail. We recommend using the same name for a adapter as the extractor it is designed for so that `scythe` can auto-detect the adapters associated with each extractor.

2.5.4 Examples of Tools Using Scythe

Materials Data Facility: https://github.com/materials-data-facility/mdf-materialsio-adapters

CHAPTER

THREE

CONTRIBUTOR GUIDE

3.1 Setting up development environment

Scythe makes use of the Poetry project to manage dependencies and packaging. To install the latest version of Scythe, first install poetry following their documentation. Once that's done, clone/download the Scythe repository locally from Github. Change into that directory and run `poetry install` (it would be a good idea to create a new virtual environment for your project first too, so as to not mix dependencies with your system environment).

By default, only a small subset of extractors will be installed (this is done so that you do not need to install all the dependencies of extractors you may never use). To install additional extractors, you can specify "extras" at install time using `poetry`. Any of the values specified in the `[tool.poetry.extras]` section of `pyproject.toml` can be provided, including `all`, which will install all bundled extractors and their dependencies. For example:

```
poetry install -E all
```

Poetry wil create a dedicated virtual environment for the project and the Scythe code will be installed in "editable" mode, so any changes you make to the code will be reflected when running tests, importing extractors, etc. It will use the default version of python available. Scythe is currently developed and tested against Python versions 3.8.12, 3.9.12, and 3.10.4. We recommend using the pyenv project to manage various python versions on your system if this does not match your system version of Python. It is required to use `tox` as well (see next paragraph). Make sure you install the versions specified in the `.python-version` file by running commands such as `pyenv install 3.8.12` etc.

Additionally, the project uses tox to simplify common tasks and to be able to run tests in isolated environments. This will be installed automatically as a development package when running the `poetry install` command above. It can be used to run the test suite with common settings, as well as building the documentation. For example, to run the full Scythe test suite on all three versions of Python targetd, just run:

```
poetry run tox
```

To build the HTML documentation (will be placed inside the `./docs/_build/` folder), run:

```
poetry run tox -e docs
```

For the sake of speed, if you would like to focus your testing on just one Python version, you can temporarily override the environment list from `pyproject.toml` with an enviornment variable. For example, to only run the test/coverage suite on Python 3.8.X, run:

```
TOXENV=py38 poetry run tox
```

Check out the `[tool.tox]` section of the `pyproject.toml` file to view how these tasks are configured, and the tox documentation on how to add your own custom tasks, if needed.

Finally, Scythe uses `flake8` to enforce code styles, which will be run for you automatically when using `tox` as defined above. Any code-style errors, such as lines longer than 100 characters, trailing whitespace, etc. will be flagged when running `poetry run tox`.

The next part of the Scythe guide details how to add a new extractor to the ecosystem.

3.2 Step 1: Implement the Extractor

Creating a new extractor is accomplished by implementing the BaseExtractor abstract class. If you are new to MaterailsIO, we recommend reviewing the User Guide first to learn about the available methods of BaseExtractor. Minimally, you need only implement the `extract`, `version`, and `implementors` operations for a new extractor. Each of these methods (and any other methods you override) must be stateless, so that running the operation does not change the behavior of the extractor.

We also have subclasses of `BaseExtractor` that are useful for common types of extractors:

- `BaseSingleFileExtractor`: Extractors that only ever evaluate a single file at a time

3.2.1 Class Attributes and Initializer

The `BaseExtractor` class supports configuration options as Python class attributes. These options are intended to define the behavior of an extractor for a particular environment (e.g., paths of required executables) or for a particular application (e.g., turning off unneeded features). We recommend limiting these options to be only JSON-serializable data types and for all to be defined in the `__init__` function to simplify text-based configuration files.

The initializer function should check if an extractor has access to all required external tools, and throw exceptions if not. For example, an extractor that relies on calling an external command-line tool should check whether the package is installed. In general, extractors should fail during initialization and not during the parsing operation if the system in misconfigured.

3.2.2 Implementing `extract`

The `extract` method contains the core logic of a Scythe extractor: rendering a summary of a group of data files. We do not specify any particular schema for the output but we do recommend best practices:

1. ***Summaries must be JSON-serializable.***
   Limiting to JSON data types ensures summaries are readable by most software without specia libraries. JSON documents are also able to be documented easily.
2. ***Human-readability is desirable.***
   JSON summaries should be understandable to users without expert-level knowledge of the data. Avoid unfamiliar acronyms, such as names of variables in a specific simulation code or settings specific to a certain brand of instrument.
3. ***Adhere closely to the original format.***
   If feasible, try to stay close to the original data format of a file or the output of a library used for parsing. Deviating from already existing formats complicates modifications to an extractor.
4. ***Always return a dictionary.***
   If an extractor can return multiple records from a single file group, return the list as an element of the dictionary. Any metadata that pertains to each of the sub-records should be stored as a distinct element rather than being duplicated in each sub-record.

We also have a recommendations for the extractor behavior:

1. ***Avoid configuration options that change only output format.***
   Extractors can take configuration options that alter the output format, but configurations should be used sparingly. A good use of configuration would be to disable complex parsing operations if unneeded. A bad use of configuration would be to change the output to match a different schema. Operations that significantly alter the form but not the content of a summary should be implemented as adaptors.
2. ***Consider whether context should be configuration.***
   Settings that are identical for each file could be better suited as configuration settings than as context.

3.2.3 Implementing `group`

The `group` operation finds all sets of files in a user-provided list files and directories that should be parsed together. Implementing `group` is optional. Implementing a new `group` method is required only when the default behavior of "each file is its own group" (i.e., the extractor only treats files individually) is incorrect.

The `group` operation should not require access to the content of the files or directories to determine groupings. Being able to determine file groups via only file names improves performance and allows for determining groups of parsable files without needing to download them from remote systems.

Files are allowed to appear in more than one group, but we recommend generating only the largest valid group of files to minimize the same metadata being generated multiple times.

It is important to note that that file groups are specific to an extractor. Groupings of files that are meaningful to one extractor need not be meaningful to another. For that reason, limit the definition of groups to sets of files that can be parsed together without consideration to what other information makes the files related (e.g., being in the same directory).

Another appropriate use of the `group` operation is to filter out files which are very unlikely to parse correctly. For example, a PDF extractor could identify only files with a ".pdf" extension. However, we recommend using filtering sparing to ensure no files are missed.

3.2.4 Implementing `citations` and `implementors`

The `citation` and `implementors` methods identify additional resources describing an extractor and provide credit to contributors. `implementors` is required, as this operation is also used to identify points-of-contact for support requests.

`citation` should return a list of BibTeX-format references.

`implementors` should return a list of people and, optionally, their contract information in the form: "FirstName LastName <email@provider.com>".

3.2.5 Implementing `version`

We require using semantic versioning for specifying the version of extractors. As the API of the extractor should remain unchanged, use versioning to indicate changes in available options or the output schema. The `version` operation should return the version of the extractor.

3.3 Step 2: Document the Extractor

The docstring for an extractor must start with a short, one sentence summary of the extractor, which will be used by our autodocumentation tooling. The rest of the documentation should describe what types of files are compatible, what context information can be used, and summarize what types of metadata are generated.

Todo: Actually write these descriptors for the available extractors

---

The Scythe project uses JSON documents as the output for all extractors and JSON Schema to describe the content of the documents. The BaseExtractor class includes a property, `schema`, that stores a description of the output format. We recommend writing your description as a separate file and having the `schema` property read and output the contents of this file. See the GenericFileExtractor source code for a example.

3.4 Step 3: Register the Extractor

3.4.1 Preferred Route: Adding the Extractor to Scythe

If your extractor has the same dependencies as existing extractors, add it to the existing module with the same dependencies.

If your extractor has new dependencies, create a new module for your extractor in `scythe`, and then add the requirements as a new key in the `[tool.poetry.extras]` section of `pyproject .toml`, following the other extractor examples in that section. Next, add your extractor to `docs/source/extractors.rst` by adding an `.. automodule::` statement that refers to your new module (again, following the existing pattern).

Scythe uses `stevedore` to simplify access to the extractors. After implementing and documenting the extractor, add it to the `[tool.poetry.plugins."scythe.extractor"]` section of the `pyproject.toml` file for Scythe. See stevedore documentation for more information (these docs reference `setup.py`, but the equivalent can be done via plugins in `pyproject .toml`; follow the existing structure if you're unsure, and ask for help from the developers if you run into issues).

3.4.2 Alternative Route: Including Extractors from Other Libraries

If an extractor would be better suited as part of a different library, you can still register it as a extractor with Scythe by altering your `pyproject.toml` file. Add an entry point with the namespace `"scythe.extractor"` and point to the class object following the stevedore documentation. Adding the entry point will let Scythe use your extractor if your library is installed in the same Python environment as Scythe.

Todo: Provide a public listing of scythe-compatible software.

So that people know where to find these external libraries

CHAPTER

FOUR

AVAILABLE EXTRACTORS

These pages detail all of the extractors currently available in Scythe.

4.1 Quick Summary

The extractors that are configured to work with the stevedore plugin are:

- ase – Parse information from atomistic simulation input files using ASE.
- crystal – Extract information about a crystal structure from many types of files.
- csv – Describe the contents of a comma-separated value (CSV) file
- dft – Extract metadata from Density Functional Theory calculation results
- em – Extract metadata specific to electron microscopy.
- filename – Extracts metadata in a filename, according to user-supplied patterns.
- generic – Gather basic file information
- image – Retrieves basic information about an image
- json – Extracts fields in JSON into a user-defined new schema.
- noop – Determine whether files exist, used for debugging
- tdb – Extract metadata from a Thermodynamic Database (TBD) file.
- xml – Extracts fields in XML into a user-defined new schema in JSON.
- yaml – Extracts fields in YAML into a user-defined new schema in JSON.

4.2 Detailed Listing

4.2.1 Generic File Extractors

Extractors that work for any kind of file

class scythe.file.**GenericFileExtractor**(*store_path=True*, *compute_hash=True*)

Gather basic file information

Parameters

- **store_path** (*bool*) – Whether to record the path of the file
- **compute_hash** (*bool*) – Whether to compute the hash of a file

4.2.2 Image Extractors

Extractors that read image data

class scythe.image.**ImageExtractor**

Retrieves basic information about an image

4.2.3 Electron Microscopy Extractors

Extractors that read electron microscopy data of various sorts (images, spectra, spectrum images, etc.) using the HyperSpy package.

class scythe.electron_microscopy.**ElectronMicroscopyExtractor**

Extract metadata specific to electron microscopy.

This parser handles any file supported by HyperSpy's I/O capabilities. Extract both the metadata interpreted by HyperSpy directly, but also any important values we can pick out manually.

For each value (if it is known), return a subdict with two keys: `value`, containing the actual value of the metadata parameter, and `unit`, a string containing a unit name from the QUDT vocabulary. Including a `unit` is optional, but highly recommended, if it is known.

The allowed metadata values are controlled by the JSONSchema specification in the `schemas/electron_microscopy.json` file.

4.2.4 Atomistic Data Extractors

Extractors related to data files that encode atom-level structure

class scythe.crystal_structure.**CrystalStructureExtractor**

Extract information about a crystal structure from many types of files.

Uses either ASE or Pymatgen on the back end

class scythe.ase.**ASEExtractor**

Parse information from atomistic simulation input files using ASE.

ASE can read many file types. These can be found at https://wiki.fysik.dtu.dk/ase/ase/io/io.html

Metadata are generated as ASE JSON DB format: https://wiki.fysik.dtu.dk/ase/ase/db/db.html

scythe.ase.**object_hook**(*dct*)

Custom decoder for ASE JSON objects

Does everything *except* reconstitute the JSON object and also converts numpy arrays to lists

Adapted from ase.io.jsonio

Parameters

dct (*dict*) – Dictionary to reconstitute to an ASE object

4.2.5 Calculation Extractors

Extractors that retrieve results from calculations

class scythe.dft.**DFTExtractor**(*quality_report=False*)

Extract metadata from Density Functional Theory calculation results

Uses the dfttopif parser to extract metadata from each file

Initialize the extractor

Parameters
quality_report (*bool*) – Whether to generate a quality report

extract(*group: Iterable[str]*, *context: dict | None = None*)

Extract metadata from a group of files

A group of files is a set of 1 or more files that describe the same object and will be used together to create a single metadata record.

Parameters

- **group** (*[str]*) – A list of one or more files that should be parsed together
- **context** (*dict*) – Context about the files

Returns
The parsed results, in JSON-serializable format.

Return type
(dict)

class scythe.ase.**ASEExtractor**

Parse information from atomistic simulation input files using ASE.

ASE can read many file types. These can be found at https://wiki.fysik.dtu.dk/ase/ase/io/io.html

Metadata are generated as ASE JSON DB format: https://wiki.fysik.dtu.dk/ase/ase/db/db.html

scythe.ase.**object_hook**(*dct*)

Custom decoder for ASE JSON objects

Does everything *except* reconstitute the JSON object and also converts numpy arrays to lists

Adapted from ase.io.jsonio

Parameters
dct (*dict*) – Dictionary to reconstitute to an ASE object

4.2.6 Structured Data Files

Extractors that read data from structured files

class scythe.csv.**CSVExtractor**(*return_records=True*, ***kwargs*)

Describe the contents of a comma-separated value (CSV) file

The context dictionary for the CSV parser includes several fields:

- `schema`: Dictionary defining the schema for this dataset, following that of FrictionlessIO
- `na_values`: Any values that should be interpreted as missing

Parameters
return_records (*bool*) – Whether to return each row in the CSV file

Keyword:
All kwargs as passed to TableSchema's infer method

citations() → List[str]

Citation(s) and reference(s) for this extractor

Returns
each element should be a string citation in BibTeX format

Return type
([str])

CHAPTER

FIVE

SCYTHE

Documentation for the non-parser functions in `scythe`.

5.1 scythe.adapters.base

Base classes for adapters

class scythe.adapters.base.**BaseAdapter**

Template for tools that transform metadata into a new form

check_compatibility(*parser:* BaseExtractor) → bool

Evaluate whether an adapter is compatible with a certain parser

Parameters
parser (BaseExtractor) – Parser to evaluate

Returns
(bool) Whether this parser is compatible

abstract transform(*metadata: dict*, *context: None | dict = None*) → Any

Process metadata into a new form

Parameters

- **metadata** (*dict*) – Metadata to transform
- **context** (*dict*) – Any context information used during transformation

Returns
Metadata in a new form, can be any type of object. `None` corresponding

version() → None | str

Version of the parser that an adapter was created for

Returns

(str) Version of parser this adapter was designed for,
or `None` if not applicable

class scythe.adapters.base.**GreedySerializeAdapter**

Converts the metadata to a string by serializing with JSON, making some (hopefully) informed choices about what to do with various types commonly seen, and otherwise reporting that the data type could not be serialized. May not work in all situations, but should cover a large number of cases.

transform(*metadata: dict*, *context=None*) → str

Process metadata into a new form

Parameters

- **metadata** (*dict*) – Metadata to transform
- **context** (*dict*) – Any context information used during transformation

Returns

Metadata in a new form, can be any type of object. None corresponding

class scythe.adapters.base.**NOOPAdapter**

Adapter that does not alter the output data

Used for testing purposes

transform(*metadata: dict*, *context=None*) → dict

Process metadata into a new form

Parameters

- **metadata** (*dict*) – Metadata to transform
- **context** (*dict*) – Any context information used during transformation

Returns

Metadata in a new form, can be any type of object. None corresponding

class scythe.adapters.base.**SerializeAdapter**

Converts the metadata to a string by serializing with JSON

transform(*metadata: dict*, *context=None*) → str

Process metadata into a new form

Parameters

- **metadata** (*dict*) – Metadata to transform
- **context** (*dict*) – Any context information used during transformation

Returns

Metadata in a new form, can be any type of object. None corresponding

5.2 scythe.utils.interface

Utilities for working with extractors from other applications

class scythe.utils.interface.**ExtractResult**(*group*, *extractor*, *metadata*)

Create new instance of ExtractResult(group, extractor, metadata)

extractor

Alias for field number 1

group

Alias for field number 0

metadata

Alias for field number 2

scythe.utils.interface.**get_adapter**(*name: str*) → *BaseAdapter*

Load an adapter

Parameters
name (*str*) – Name of adapter

Returns
(BaseAdapter) Requested adapter

scythe.utils.interface.**get_available_adapters**() → dict

Get information on all available adapters

Returns
(dict) Where keys are adapter names and values are descriptions

scythe.utils.interface.**get_available_extractors**()

Get information about the available extractors

Returns
Descriptions of available extractors

Return type
[dict]

scythe.utils.interface.**get_extractor**(*name: str*) → *BaseExtractor*

Load an extractor object

Parameters
name (*str*) – Name of extractor

Returns
Requested extractor

scythe.utils.interface.**get_extractor_and_adapter_contexts**(*name*, *global_context*, *extractor_context*, *adapter_context*)

Helper function to update the helper and adapter contexts and the 'name'
of a extractor/adapter pair

Parameters

- **name** (*str*) – adapter/extractor name.
- **global_context** (*dict*) – Context of the files, used for every extractor and adapter
- **adapter_context** (*dict*) – Context used for adapters. Key is the name of the adapter, value is the context. The key `@all` is used to for context used for every adapter
- **extractor_context** (*dict*) – Context used for adapters. Key is the name of the extractor, value is the context. The key `@all` is used to for context used for every extractor

Returns
extractor_context, my_adapter context tuple

Return type
(dict, dict)

scythe.utils.interface.**run_all_extractors_on_directory**(*directory: str*, *global_context=None*, *adapter_context: None | dict = None*, *extractor_context: None | dict = None*, *include_extractors: None | List[str] = None*, *exclude_extractors: None | List = None*, *adapter_map: None | str | Dict[str, str] = None*, *default_adapter: None | str = None*) → Iterator[*ExtractResult*]

Run all known files on a directory of files

Parameters

- **directory** (*str*) – Path to directory to be parsed
- **global_context** (*dict*) – Context of the files, used for every extractor and adapter
- **adapter_context** (*dict*) – Context used for adapters. Key is the name of the adapter, value is the context. The key `@all` is used to for context used for every adapter
- **extractor_context** (*dict*) – Context used for adapters. Key is the name of the extractor, value is the context. The key `@all` is used to for context used for every extractor
- **include_extractors** (*[str]*) – Predefined list of extractors to run. Only these will be used. Mutually exclusive with *exclude_extractors*.
- **exclude_extractors** (*[str]*) – List of extractors to exclude. Mutually exclusive with *include_extractors*.
- **adapter_map** (*str, dict*) – Map of extractor name to the desired adapter. Use 'match' to find adapters with the same names
- **default_adapter** (*str*) – Adapter to use if no other adapter is defined

Yields

((str), str, dict) Tuple of (1) group of files, (2) name of extractor, (3) metadata

scythe.utils.interface.**run_all_extractors_on_group**(*group*, *adapter_map=None*, *global_context=None*, *adapter_context: None | dict = None*, *extractor_context: None | dict = None*, *include_extractors: None | List[str] = None*, *exclude_extractors: None | List = None*, *default_adapter: None | str = None*)

Parse metadata from a file-group and adapt its metadata per a user-supplied adapter_map.

This function is effectively a wrapper to execute_extractor() that enables us to output metadata in the same format as run_all_extractors_on_directory(), but just on a single file group.

Parameters

- **group** (*[str]*) – Paths to group of files to be parsed
- **global_context** (*dict*) – Context of the files, used for every extractor and adapter
- **adapter_context** (*dict*) – Context used for adapters. Key is the name of the adapter, value is the context. The key `@all` is used to for context used for every adapter
- **extractor_context** (*dict*) – Context used for adapters. Key is the name of the extractor, value is the context. The key `@all` is used to for context used for every extractor
- **include_extractors** (*[str]*) – Predefined list of extractors to run. Only these will be used. Mutually exclusive with *exclude_extractors*.

- **exclude_extractors** (*[str]*) – List of extractors to exclude. Mutually exclusive with *include_extractors*.
- **adapter_map** (*str, dict*) – Map of extractor name to the desired adapter. Use 'match' to find adapters with the same names:
- **default_adapter** –

Yields
Metadata for a certain

scythe.utils.interface.**run_extractor**(*name*, *group*, *context=None*, *adapter=None*)

Invoke a extractor on a certain group of files

Parameters

- **name** (*str*) – Name of the extractor
- **group** (*[str]*) – Paths to group of files to be parsed
- **context** (*dict*) – Context of the files, used in adapter and extractor
- **adapter** (*str*) – Name of adapter to use to transform metadata

Returns
Metadata generated by the extractor

Return type
([dict])

5.3 scythe.utils.grouping

Utilities for implementing grouping operations

scythe.utils.grouping.**group_by_postfix**(*files: Iterable[str]*, *vocabulary: List[str]*) → Iterable[Tuple[str, ...]]

Group files that have a common ending

Finds all filenames that begin with a prefixes from a user-provided vocabulary and end with the same post-fix.

For example, consider a directory that contains files A.1, B.1, A.2, B.2, and C.1. If a user provides a vocabulary of ['A', 'B'], the parser will return groups (A.1, B.1) and (A.2, B.2). If a user provides a vocabulary of ['A', 'B', 'C'], the parser will return groups (A.1, B.1), (A.2, B.2), and (C.1)

See `scythe.dft.DFTParser` for an example usage.

Parameters

- **files** (*[str]*) – List of files to be grouped
- **vocabulary** (*[str]*) – List of known starts for the file

Yields
([str]) – Groups of files to be parsed together

scythe.utils.grouping.**preprocess_paths**(*paths: str | Path | List[str] | List[Path]*) → List[str]

Transform paths to absolute paths

Designed to be used to simplify grouping logic

Parameters
paths (*Union[str, List[str]*) – Files and directories to be parsed

Returns
List of paths in standardized form

Return type
(List[str])

PYTHON MODULE INDEX

S

INDEX

T

V