Code guidelines

NOMAD has a long history and many people are involved in its development. These guidelines are set out to keep the code quality high and consistent. Please read them carefully.

Principles and rules

- simple first, complicated only when necessary
- search and adopt generic established 3rd-party solutions before implementing specific solutions
- only unidirectional dependencies between components/modules, no circles
- only one language: Python (except GUI of course)

The are some *rules* or better strong *guidelines* for writing code. The following applies to all Python code (and where applicable, also to Javascript and other code):

- Use an IDE (e.g. VS Code (https://code.visualstudio.com/){:target="_blank"}) or otherwise automatically enforce code formatting and linting (https://code.visualstudio.com/docs/python/linting){:target="_blank"}.
- Use `nomad qa` before committing. This will run all tests, static type checks, linting, etc.
- Test the public interface of each submodule (i.e. Python file).
- There is a style guide to Python. Write PEP 8 (https://www.python.org/dev/peps/pep-0008/){:target="_blank"}-compliant Python code. An exception is the line cap at 79, which can be broken but keep it 90-ish.
- Be Pythonic (https://docs.python-guide.org/writing/style/){:target="_blank"} and watch

this talk about best practices (https://www.youtube.com/watch?v=wf-BqAjZb8M){:target="_blank"}.

- Add docstrings to the *public* interface of each submodule (i.e. Python file). This includes APIs that are exposed to other submodules (i.e. other Python files).
- The project structure follows this guide (https://docs.python-guide.org/writing/structure/){:target="_blank"}. Keep it!
- Write tests for all contributions.
- Adopt *Clean Code* practices. Here is a good introductory talk to Clean Code (https://youtu.be/7EmboKQH8lM){:target="_blank"}.

Enforcing rules with CI/CD

These *guidelines* are partially enforced by CI/CD. As part of CI all tests are run on all branches; further we run a *linter*, *PEP 8* checker, and *mypy* (static type checker). You can run `nomad qa` to run all these tests and checks before committing.

See the contributing guide (../howto/develop/contrib.md) for more details on how to work with issues,
branches, merge requests, and CI/CD.

Documenting code

Write Clean Code (https://youtu.be/7EmboKQH8lM){:target="_blank"} that is easy to comprehend.

However, you should document the whole publicly exposed interface of a module. For Python
this includes most classes and functions that you will write, for React its exported components and their props.

For all functionality that is exposed to clients (APIs, CLI, schema base classes and annotations, UI functionality), you must consider to add explanations, tutorials, and examples to the documentation system (i.e. the `docs` folder). This is built with mkdocs (https://www.mkdocs.org/){:target="_blank"} and published as part of each NOMAD installation.

Also mind `nomad/mkdocs.py` and `mkdocs.yaml` and have a look at used plugins and extra
functions, e.g. this includes generation of Markdown from `examples` or Pydantic models.

To document Python functions and classes, use Google
docstrings
(https://github.com/NilsJPWerner/autoDocstring/blob/HEAD/docs/google.md){:target="_blank"}.

Use Markdown if you need to add markup but try to reduce this to a minimum.

You can use VS Code plugins like
autoDocstring
(https://github.com/NilsJPWerner/autoDocstring/tree/f7bc9f427d5ebcd87e6f5839077a87ecd1cbb404){:target="_blank"}
to help.

Always use single quotes, pad single-line docstrings with spaces and start multi-line ones
on a new line.

Here are a few examples:

```python
def generate_uuid() -> str:
    '''Generates a base64 encoded Version 4 unique identifier. '''
    return base64.encode(uuid4())

def add(a: float, b: float) -> float:
    '''
    Adds two numbers.
    Args:
      a (float): One number.
      b (float): The other number.
    Returns:
      float: The sum of a and b.
    '''
    return a + b
```

The only reason to comment individual lines is because there is absolutely no way to write it simple enough. The typical scenarios are:

- workarounds to known issues with used dependencies
- complex interactions between seemingly unrelated pieces of code that cannot be resolved
  otherwise
- code that has to be cumbersome due to performance optimizations

Do not comment out code. We have Git for that.

Names and identifiers

There is a certain terminology consistently used in this documentation and the source code. Use this terminology for identifiers.

Do not use abbreviations. There are (few) exceptions: `proc` (processing), `exc` or `e` (exception), `calc` (calculation), `repo` (repository), `utils` (utilities), and `aux` (auxiliary).

Other exceptions are `f` for file-like streams and `i` for index running variables, although the latter is almost never necessary in Python.

Terms:

- *upload*: A logical unit that comprises a collection of files uploaded by a user, organized in a directory structure.
- *entry*: An archive item, created by parsing a *mainfile*. Each entry belongs to an upload and is associated with various metadata (an upload may have many entries).
- *child entry*: Some parsers generate multiple entries -- a *main* entry plus some number of *child* entries. Child entries are identified by the *mainfile* plus a *mainfile_key* (string value).
- *calculation*: denotes the results of either a theoretical computation created by CMS code, or an experiment.
- *raw file*: A user uploaded file, located somewhere in the upload's directory structure.
- *mainfile*: A raw file identified as parsable, defining an entry of the upload in question.
- *aux file*: Additional files within an upload.
- *entry metadata*: Some quantities of an entry that are searchable in NOMAD.
- *archive data*: The normalized data of an entry in NOMAD's Metainfo-based format.

Throughout NOMAD, we use different ids. If something is called *id*, it is usually a random uuid and has no semantic connection to the entity it identifies. If something is called a *hash* then it is a hash generated based on the entity it identifies. This means either the whole thing or just some properties of this entities.

- The most common hash is the `entry_hash` based on `mainfile` and aux file contents.
- The `upload_id` is a UUID assigned to the upload on creation. It never changes.
- The `mainfile` is a path within an upload that points to a file identified as parsable. This also uniquely identifies an entry within the upload.
- The `entry_id` (previously called `calc_id`) uniquely identifies an entry. It is a hash

over the `mainfile` and respective `upload_id`. **NOTE:** For backward compatibility, `calc_id` is also still supported in the API, but using it is strongly discouraged.
- We often use pairs of `upload_id/entry_id`, which in many contexts allow to resolve an entry-related file on the filesystem without having to ask a database about it.
- The `pid` or (`coe_calc_id`) is a legacy sequential integer id, previously used to identify entries. We still store the `pid` on these older entries for historical purposes.
- Calculation `handle` or `handle_id` are created based on those `pid`.
  To create hashes we use :py:func:`nomad.utils.hash`.

Logging

There are three important prerequisites to understand about nomad-FAIRDI's logging:
- All log entries are recorded in a central Elasticsearch database. To make this database useful, log entries must be sensible in size, frequency, meaning, level, and logger name. Therefore, we need to follow some rules when it comes to logging.
- We use a *structured* logging approach. Instead of encoding all kinds of information in log messages, we use key-value pairs that provide context to a log *event*. In the end, all entries are stored as JSON dictionaries with `@timestamp`, `level`, `logger_name`, `event` plus custom context data. Keep events very short, most information goes into the context.
- We use logging to inform about the state of nomad-FAIRDI, not about user behavior, input, or data. Do not confuse this when determining the log level for an event.
  For example, a user providing an invalid upload file should never be an error.

Please follow the following rules when logging:
- If a logger is not already provided, only use :py:func:`nomad.utils.get_logger` to acquire a new logger. Never use the built-in logging directly. These loggers work like the system loggers, but allow you to pass keyword arguments with additional context

data. See also the structlog docs (https://structlog.readthedocs.io/en/stable/){:target="_blank"}.

- In many context, a logger is already provided (e.g. API, processing, parser, normalizer). This provided logger has already context information bounded. So it is important to use those instead of acquiring your own loggers. Have a look for methods called `get_logger` or attributes called `logger`.
- Keep events (what usually is called *message*) very short. Examples are: *file uploaded*, *extraction failed*, etc.
- Structure the keys for context information. When you analyze logs in ELK, you will see that the set of all keys over all log entries can be quite large. Structure your keys to make navigation easier. Use keys like `nomad.proc.parser_version` instead of `parser_version`. Use module names as prefixes.
- Don't log everything. Try to anticipate how you would use the logs in case of bugs, error scenarios, etc.
- Don't log sensitive data.
- Think before logging data (especially dicts, list, NumPy arrays, etc.).
- Logs should not be abused as a *printf*-style debugging tool.

The following keys are used in the final logs that are piped to Logstash.

Notice that the key name is automatically formed by a separate formatter and may differ

from the one used in the actual log call.

Keys that are autogenerated for all logs:

- `@timestamp`: Timestamp for the log
- `@version`: Version of the logger
- `host`: Host name from which the log originated
- `path`: Path of the module from which the log was created

- `tags`: Tags for this log
- `type`: *message_type* as set in the LogstashFormatter
- `level`: Log level: `DEBUG`, `INFO`, `WARNING`, `ERROR`
- `logger_name`: Name of the logger
- `nomad.service`: Service name as configured in `config.py`
- `nomad.release`: Release name as configured in `config.py`

Keys that are present for events related to processing an entry:

- `nomad.upload_id`: id of the currently processed upload
- `nomad.entry_id`: id of the currently processed entry
- `nomad.mainfile`: mainfile of the currently processed entry

Keys that are present for events related to exceptions:

- `exc_info`: Stores the full Python exception that was encountered. All uncaught
  exceptions will be stored automatically here.
- `digest`: If an exception was raised, the last 256 characters of the message are stored
  automatically into this key. If you wish to search for exceptions in
  Kibana (https://www.elastic.co/de/kibana){:target="_blank"}, you will want to use this value as it will
  be indexed unlike the full exception object.

Copyright notices

We follow this
recommendation of the Linux Foundation (https://www.linuxfoundation.org/blog/2020/01/copyright-notices-in-open-source-software-projects/){:target="_blank"}
for the copyright notice that is placed on top of each source code file.

It is intended to provide a broad generic statement that allows all authors/contributors of the NOMAD project to claim their copyright, independent of their organization or individual ownership.

You can simply copy the notice from another file. From time to time we can use a tool like licenseheaders (https://pypi.org/project/licenseheaders/){:target="_blank"} to ensure correct
notices. In addition we keep a purely informative AUTHORS file.

Git submodules and other "in-house" dependencies

As the NOMAD ecosystem grows, you might develop libraries that are used by NOMAD instead
of being part of its main codebase. The same guidelines should apply. You can use
GitHub Actions (https://github.com/features/actions){:target="_blank"} if your library is hosted on Github
to ensure automated linting and tests.