# The SENTINEL archive data format

***Release 0.1.dev0***

**SENTINEL collaboration**

**Jan 28, 2021**

## Contents



Our objectives are to develop a standard model and data specification through which the models in SENTINEL can be linked together, including inputs and outputs, and to classify the models according to this specification.

The primary purpose of the interface framework is to allow data to flow between very different types of models, in a way that can be automated. This is challenging because not only do different models use different data formats, but also, metadata may or may not exist, and conventions on units or variable naming differ between different models and research communities.

To address this, we develop a flexible data standard that can form the glue between different models, allowing them to be used together, and a user interface that makes it easier for modelling teams to actually make use of this data format. All software described here is available as open source software under the version 2 of the Apache software license.

## 1 The SENTINEL archive data format

The SENTINEL partners work with a variety of models in diverse computing environments. The **SENTINEL archive** data format[1] has been chosen to facilitate interoperability between these diverse modelling frameworks by making sharing of data easier. Since we have to cope with a variety of research requirements, and workflows, the format is fairly flexible, and to a great extent self-descriptive. We rely on the author of a dataset to also describe it. A typical description of a dataset includes both metadata and structural information.

Metadata of a dataset typically establishes the context for the dataset. It can consist of properties like:

- a computer program friendly nameso that they can be referred to easily from software,

[1] We use the term "data format" to refer to the general structure of the data, and its metadata, instead of a specific file type like *CSV*, *Excel*, etc.

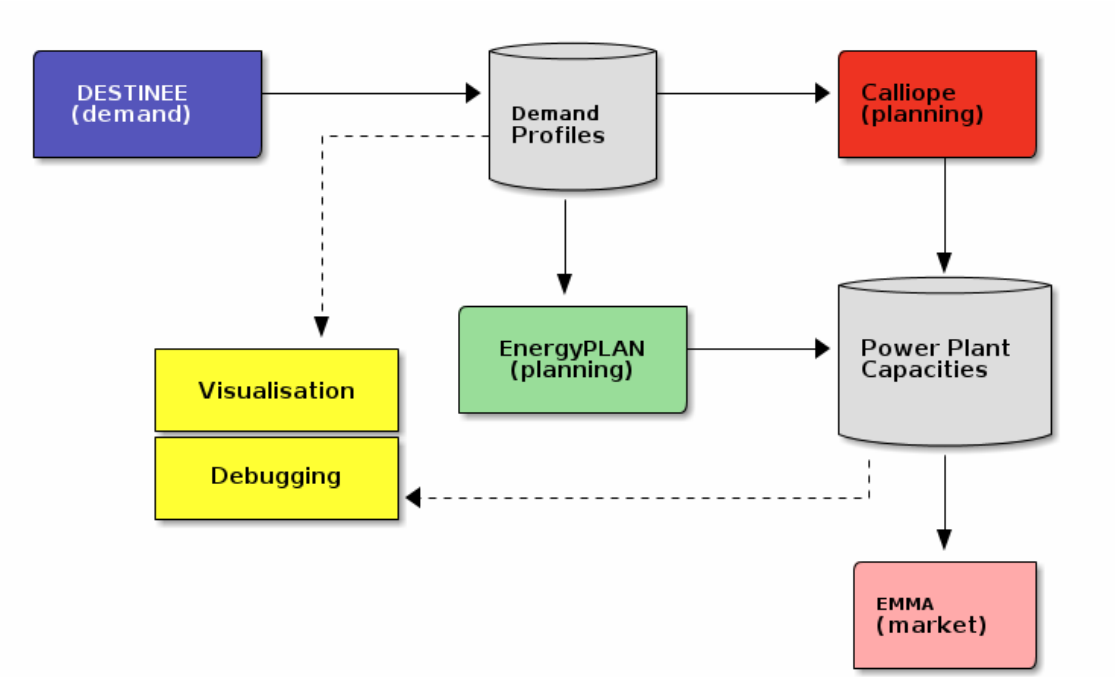


Fig. 1: An open ecosystem enabled by the data format, where different modelling frameworks can be interlinked, and a suit of independent software tools to work with data can be developed.

- a title and free-form descriptive text so that other researchers using the dataset are aware of its provenance and use it correctly,
- search keywords for easier discoverability on online platforms,
- license information so that others know the terms of use of the data, and
- citation information.

Whereas structural information should describe the type information, any contraints, and assumptions implicit in the data. Our implementation builds on top of the *frictionless datapackage* specification.

## 1.1 What is a `datapackage`?

A datapackage consists of a set of data files, any related source code, relevant licenses, and the `datapackage.json` file, that records all this information in a single place. It is based on a widely recognised standard called *frictionless data*; further detals of the specification can be found on their website. This file also includes specific information about the structure of each dataset (or *data resource*). A data resource can be any kind of file, like *CSV*, *Excel*, etc. At the moment, only tabular resources are supported. Each data resource has an entry which states its name, relative path, and structure (or *schema*) of the data within it. Structural information includes column names, the type of data stored in each column, instructions on how to identify missing values, or how to uniquely identify each row in a dataset (otherwise known as "primary key"). Since data can often be large, it is possible to split the schema into a separate file and include it from `datapackage.json` so as to keep it manageable and easy to work with.

We provide a web-based user interface (web UI) and a Python API to create datasets. The web UI is meant to be easy to use, and requires no programming knowledge, whereas the Python API is more fully featured and requires some understanding of the scientific Python ecosystem. At the moment there is no direct support for other languages, but the frictionless data specification provides some alternate implementations of the datapackage format which maybe

used. However, since the underlying design uses commonly used facilities, e.g. using *JSON* for metadata, and relying on well established file formats; adding support in other languages is a matter of allocating resources.

# 2 Using a `datapackage` from Python

The Python libraryprovides a convenient way to create, modify, and read datapackages. Some of the API is described below with code examples for illustration.

## 2.1 Metadata

To create a `datapackage.json` file, you need not write it from scratch. A datapackage can be created within Python with the following:

```
from sark.dpkg import create_pkg
from sark.metatools import get_license

pkg_meta = {
    "name": "dataset_for_xyz",
    "title": "Dataset for XYZ",
    "description": "This dataset is for XYZ and spans 10 years",
    "licenses": [get_license("CC0-1.0")],
    "keywords": ["XYZ", "ABC"],
}
pkg = create_pkg(pkg_meta, Path("data").glob("*.csv"))
```

In the above snippet the dictionary `pkg_meta` sets the metadata, whereas all *CSV* files in the subdirectory `data/` are added to the datapackage as data resources. As the files are read, a basic *schema* is guessed. You may inspect the contents of the datapackage by looking at `pkg.descriptor`.

## 2.2 Schema

The schema can be specialised further by updating the metadata for each field. To illustrate with an example, say we have a time series dataset called "electricity-consumption", and the first column in the file contains timestamps, or `datetime` values. The heuristics that infers the schema detects the column as a `string` (plain text). To rectify this, we can use the snippet below:

```
from sark.dpkg import update_pkg

update_fields = {
    "time": {"name": "time", "type": "datetime", "format": "default"},
}
success = update_pkg(pkg, "electricity-consumption"), update_fields)

if success:
    print(f"successfully updated: {list(update_fields)}")
else:
    print(f"failed to update: {list(update_fields)}")
```

If multiple fields need updating, you only need to add a key corresponding to the field in the `update_fields` dictionary; the following snippet also updates the "QWE" column such that it is interpreted as `numbers`:

```
update_fields = {
    "time": {"name": "time", "type": "datetime", "format": "default"},
    "QWE": {"name": "QWE", "type": "integer", "format": "default"},
}
```

### 2.2.1 Missing values & primary keys

If for instance there is a need to add to the default list of values that are treated as missing values, or to specify a set of fields as primary keys, use the following:

```
update = {
    "primaryKey": ["lvl", "TRE", "IUY"],
    "missingValues": ["", "nodata"]
}
success = update_pkg(pkg, "electricity-consumption"), update, fields=False)
```

In the above example, the fields: "lvl", "TRE", and "IUY", are specified as primary keys (used to uniquely identify a row in a dataset). Similarly, the token `nodata` is being added to the list of values to be treated as a missing value.

## 2.3 Persistence

Once a datapackage has been created, it can be saved to disk as a *JSON* file, typically named `datapackage.json`. Alongside this, a datapackage consists of other resources like the data files, and other resources referred in the package description. These other resources need to be at the same relative path, otherwise we will not be able to read the datapackage correctly. So for sharing datapackages, we either have the option of sharing them as public repositories, preserving the directory hierarchy, or bundle everything in archive files like *ZIP*. You can use the `write_pkg` function to do this.

```
from sark.dpkg import write_pkg

# create package: `pkg`
write_pkg(pkg, "path/to/datapackage.json")
# or
write_pkg(pkg, "path/to/mydatapackage.zip")
```

The *ZIP* archive preserves the relative directory hierarchy of the datapackage, as shown in the example below:

```
$ unzip -l /tmp/mydatapackage.zip
Archive:  /tmp/mydatapackage.zip
  Length      Date    Time    Name
---------  ---------- -----   ----
    14013  05-07-2020 17:00   data/sample-ok-1.csv
    15446  05-07-2020 17:00   data/sample-ok-2.csv
     1741  05-18-2020 17:23   datapackage.json
---------                     -------
    31200                     3 files
```

## 2.4 Reading as a `pandas.DataFrame`

A datapackage can be read into Python by using the underlying `datapackage` library. However it is not very performant, and does not integrate with the popular data analysis library `pandas`. That is why the SENTINEL archive library provides a helper function that reads the package schema, and reads a data resource directly into a `pandas.DataFrame`.

```
from sark.dpkg import to_df

pkg = ... # read datapackage from disk

# read all data resources as a dataframe
dfs = [to_df(resource) for resource in pkg.resources]
```

Depending on the type specified in the schema, a field is converted to a `pandas` type. The mapping between the two kinds of types are summarised below:

| schema type | `pandas` type |
|---|---|
| `boolean` | `bool` |
| `datetime` | `datetime64` |
| `integer` | `Int64` |
| `number` | `float` |
| `string` | `string` |

Note the choice of `Int64` and `string` among the types. This adds a minimum requirement of Pandas version 1.0. `Int64` allows you to have missing values in integer columns, which would be otherwise impossible; and using `string` is much more space efficient, making it easier to manage larger datasets with lots of text fields.