opensanctions

Release 3.1.1

OpenSanctions Team

Oct 03, 2021

CONTENTS:

This technical documentation is intended to be read by Python developers who wish to run the OpenSanctions crawlers on their own infrastructure, or plan to add their own crawlers to the system.

For users who merely want to consume the datasets produced by the project, we suggest you browse the project home page instead.

CHAPTER

ONE

SYSTEM OVERVIEW

The OpenSanctions pipeline handles the following key steps:

- Fetching entity data from online sources and parsing the original format
- Normalising the entities in source data into the *OpenSanctions data model*
- Storing entities to a local staging database for processing
- Merging entities across different *sources* (TODO)
- Exporting data into a variety of target formats (JSON, CSV)

These steps are triggered using a command-line utility, `opensanctions`, which can run *parts or all of this process* for specific segments of the data.

1.1 Installation

The OpenSanctions data pipeline can be installed in a few different ways, depending on your answers to these two questions:

- Do you just want to execute the existing crawlers, or change them and add your own data sources to the system?
- Are you more comfortable running the program in your own Python virtual environment, or do you prefer to isolate it in a Docker container?

While getting OpenSanctions to run inside a Docker container is very easy, it might make iteration a bit slower and stand in the way of debugging a crawler as it is being developed.

In any case, you will need to check out the OpenSanctions application from its repository to your computer:

```
$ git clone https://github.com/pudo/opensanctions.git
$ cd opensanctions
```

The steps below assume you're working within a checkout of that repository.

1.1.1 Using Docker

If you have Docker installed on your computer, you can use the supplied `Makefile` and `docker-compose` configuration to build and run a container that hosts the application:

```
$ make build
# This runs a single command which you can also execute directly:
$ docker-compose build --pull
```

Once the container images have been built, you can run the `opensanctions` command-line tool within the container:

```
$ docker-compose run --rm app opensanctions --help
# Or, run a specific subcommand:
$ docker-compose run --rm app opensanctions crawl eu_fsf
# You can also just run a shell inside the container, and then execute multiple
# commands in sequence:
$ docker-compose run --rm app bash
container$ opensanctions crawl eu_fsf
# The above command to spawn an interactive shell is also available as:
$ make shell
```

The docker environment will provide the commands inside the container with access to the `data/` directory in the current working directory, i.e. the repository root. You can find any generated outputs and the copy of the processing database in that directory.

1.1.2 Python virtual environment

OpenSanctions functions as a fairly stand-alone Python application, albeit with a large number of library dependencies. That's why we'd suggest that you should never install OpenSanctions directly into your system Python, and instead always use a virtual environment. Within a fresh virtual environment (Python >= 3.9), you should be able to install OpenSanctions using `pip`:

```
# Inside the opensanctions repository path:
$ pip install -e .
# You can check if the application has been installed successfully by
# invoking the command-line tool:
$ opensanctions --help
```

If you encounter any errors during the installation, please consider googling errors related to libraries used by OpenSanctions (e.g.: SQLAlchemy, Python-Levenshtein, click, etc.).

Please avoid installing the `opensanctions` package from PyPI via pip. The package exists mainly to claim the package name but is not regularly updated as part of the build/release process of OpenSanctions.

Hint: OpenSanctions has an optional dependency on PyICU, a library related to the transliteration of names in other alphabets to the latin character set. This library is not installed by default because its configuration can be tricky.

Consider following the PyICU documentation to install this library and achieve better transliteration results.

1.1.3 Configuration

OpenSanctions is inspired by the twelve factor model and uses environment variables to configure the operation of the system. Some of the key settings include:

- `OPENSANCTIONS_DATABASE_URI` is a database connection string, such as `sqlite:///filename.sqlite` or `postgresql://user:pass@host/database`. Only PostgreSQL and SQLite are supported as backends.
- `OPENSANCTIONS_DATA_PATH` is the main working directory for the system. By default it will contain cached artifacts and the generated output data. This defaults to the `data/` subdirectory of the current working directory when the `opensanctions` command is invoked.
- `OPENSANCTIONS_METADATA_PATH` is the path in which the system will search for metadata specifications of *datasets*. By default, this points to the `metadata/` subdirectory within the application source code.

1.2 Using the system

Below you'll find instructions on how to run the OpenSanctions software and how to add additional crawlers to the system. Before we dive into that, however, let's explore some of the concepts underlying the system.

1.2.1 Datasets, sources and collections

OpenSanctions collects data from a variety of sanctions lists and other data providers and converts it into a common, simple-to-use data model. These data are grouped into **datasets**. Some datasets are **sources** and refer to a data origin (e.g. `eu_fsf`, the EU sanctions list. Other datasets combine data from multiple sources into a **collection** (e.g. `sanctions`, which collects all sanctions entities from multiple sources).

Both **source** and **collection datasets** have a metadata definition, stored as a YAML file in `opensanctions/metadata`. **Sources** also include crawler code to parse and import the material. This code is usually located in `opensanctions/crawlers`.

1.2.2 Entities and targets

The main objective of OpenSanctions is to combine data from multiple sources into a common data model. To this end, the system uses FollowTheMoney (FtM), a data modelling and validation library which defines a set of entity schemata, such as Person, Company, Address or Sanction. FtM-based entities are stored in a local database and then exported to a variety of file formats.

A peculiarity of the data in OpenSanctions is that sources may mention entities that are merely adjacent to a sanctions target, but not themselves sanctioned. To distinguish the sanctioned entities, they are flagged as **targets** in the database. For most end users that wish to download and use a simple CSV file, chances are that they will want sanctions targets, without the secondary entities in the dataset.

1.2.3 Using the command-line tool

Once you've successfully *installed* OpenSanctions, you can use the built-in command-line tool to run parts of the system:

```
# Crawl and export the US consolidated list:
$ opensanctions run us_ofac_cons

# This works for both sources and collections. Running a collection will
# crawl all related sources and then export the collection data:
$ opensanctions run sanctions

# Running without a specified dataset name will default to using the
# `all` collection which contains all sources:
$ opensanctions run
$ opensanctions run all

# If you're developing the crawler, you can skip generating the exports and
# only run the crawl stage:
$ opensanctions crawl us_ofac_cons

# Inversely, you can also export a dataset without re-crawling the sources:
$ opensanctions export us_ofac_cons

# During development you might also want to force delete all data linked
# to a source:
$ opensanctions clear us_ofac_cons
```

The available dataset names are determined from the set of metadata YAML files found in `OPENSANCTIONS_METADATA_PATH` (see: *configuration*).

1.3 Developing a crawler

Note: Please consult the contribution guidelines before developing new crawlers to learn about inclusion criteria for new data sources.

A crawler is a small Python script that will import data from a web origin and store it as entities as a *source dataset*. The basic process for creating a new crawler is as follows:

1. File a GitHub issue to discuss the suggested source
2. Create a YAML metadata description for the new source
3. Create a Python script to fetch and process the data
4. Address any data normalisation issues the framework might report

In the future, an additional step will be required to link up duplicate entities against other sources and to define canonical/merged entities.

1.3.1 Source metadata

Before programming a crawler script, you need to create a YAML file with some basic metadata to describe the new dataset. That information includes the dataset name (which is normally derived from the YAML file name), information about the source publisher and the source data URL.

The metadata file must also include a reference to the *entry point*, the Python code that should be executed in order to crawl the source.

Create a new YAML file in the path `opensanctions/metadata` named after your new dataset. By convention, a dataset name should start with the ISO 3166-2 code of the country it relates to, and name parts should be separated by underscores. The contents of the new metadata file should look like this:

> **Warning:** The dataset metadata format is going to be subject to significant change.

```yaml
title: "Financial Sanctions Files (FSF)"
url: https://eeas.europa.eu/

# The description should be extensive, and can use markdown for formatting:
description: >
   As part of the Common Foreign Security Policy thr European Union publishes
   a sanctions list that is implemented by all member states.

# The Python module that contains the crawler code:
entry_point: opensanctions.crawlers.eu_fsf_demo

# A prefix will be used to mint entity IDs. Keep it short.
prefix: eu-fsf

# Define what collections the source is part of. All sources are added to a
# magical collection called 'all'. Each collection also has its own YAML
# metadata file, but the link between a source and the collections it is a
# part of is established via the source metadata, not the collection metadata.
collections:
   - sanctions

# This section provides information about the original publisher of the data,
# often a government authority:
publisher:
   organization: European Commission
   authority: European Union External Action Service
   acronym: EEAS
   country: eu
   url: https://eeas.europa.eu/topics/sanctions-policy/8442/consolidated-list-of-
↪sanctions_en

# Information about the data, including a deep link to a downloadable file, if
# one exists.
data:
   url: https://webgate.ec.europa.eu/europeaid/fsd/fsf/public/files/xmlFullSanctionsList_
↪1_1/content
   format: XML
```

Once that YAML file is stored in the correct folder, you should be able to run command-line operations against the dataset, for example (if your metadata file is named *eu_fsf_demo.yml*):

```
$ opensanctions run eu_fsf_demo
....
ModuleNotFoundError: No module named 'opensanctions.crawlers.eu_fsf_demo'
```

That error will be addressed in the next section, by adding a crawler script.

1.3.2 Developing a crawler script

In order to actually feed data into the data source, we need to write a crawler script. The script location is specified in the YAML metadata file as `entry_point:`. This also means you could reference the same script for multiple data sources, for example in a scenario where two data sources use the API, except with some varied parameters.

In our example above, we'd create a file in `opensanctions/crawlers/eu_fsf_demo.py` with a crawler skeleton:

```
def crawl(context):
    context.log.info("Hello, World!")
```

Running the crawler (`opensanctions crawl eu_fsf_demo`) should now produce a log line with the message *Hello, World!*

You'll notice that the `crawl()` function receives a `context` object. Think of it as a sort of sidekick: it helps you to create, store and document data in your crawler.

Fetching and storing resources

Many crawlers will start off by downloading a source data file, like a CSV table or a XML document. The `context` provides utility methods that let you fetch a file and store it into the crawlers working directory. Files stored to the crawler home directory (`context.path`) will later be uploaded and published to the web.

```
def crawl(context):
    # Fetch the source data URL specified in the metadata to a local path:
    source_path = context.fetch_resource('source.xml', context.dataset.data.url)
    with open(source_path, 'r') as fh:
        print(len(fh.read()))

    # You can also register the file as a resource with the dataset that
    # will be included in the exported metadata index:
    context.export_resource(source_path, title="Source data XML file")
```

Other crawlers might not be as lucky: instead of fetching their source data as a single bulk file, they might need to crawl a large number of web pages to collect the necessary data. For this, access to a pre-configured Python `requests` session object is provided:

```
from lxml import html

def crawl(context):
    response = context.http.get(context.dataset.data.url)

    # Parse the HTTP response into an lxml DOM:
    doc = html.fromstring(response.text)
```

(continues on next page)

(continued from previous page)

```
    # Query the DOM for specific elements to extract data from:
    for element in doc.findall('.//div[@class="person"]'):
        context.log.info("Element", element=element)
```

Responses from the `context.http` session are cached between different runs of the crawler and will be cached for up to 10 days. You can partially disable this by adding a timestamp parameter to the fetched URLs.

Creating and emitting entities

The goal of each crawler is to produce data about *persons and other entities of interest*. To enable this, the `context` provides a number of helpers that construct and store *entities*:

```
def crawl(context):

    # Create an entity object to which other information can be assigned:
    entity = context.make("Person")

    # Each entity needs a unique ID. In OpenSanctions, this is often derived
    # from the ID of a source database, or a string:
    entity.make_slug('Joseph Biden')

    # Assign some property values:
    entity.add('name', 'Joseph Robinette Biden Jr.')
    entity.add('alias', 'Joe Biden')
    entity.add('birthDate', '1942-11-20')

    # Invalid property values ('never' is not a date) will produce a log
    # error:
    entity.add('deathDate', 'never')

    # Store or update the entity in the database:
    context.emit(entity, target=True)
```

The entity object is based on the entity proxy in FollowTheMoney, so we suggest you also check out the FtM documentation on entity construction. Some additional utility methods are added in the `entity class` in OpenSanctions.

1.3.3 Checklist

When contributing a new data source, or some other change, make sure of the following:

- You've created a metadata YAML file with detailed descriptions and links to the source URL.
- Your code should run after doing a simple `pip install` of the codebase. Include additional dependencies in the `setup.py`. Don't use non-Python dependencies like `Headless Chrome` or `Selenium`.
- The output data for your crawler should be Follow The Money objects. If you need more fields added to the ontology, submit a pull request upstream. Don't include left-over data in an improvised way.
- Include verbose logging in your crawler. Make sure that new fields or enum values introduced upstream (e.g. a new country code or sanction program) will cause a warning to be emitted.
- Bonus points: your Python code is linted and formatted with `black`.

1.3.4 Using the context

1.3.5 Type lookups

TODO

1.4 Crawler helpers

The helpers module contains a large number of functions that help you process source data into the correct shape for the common OpenSanctions data format.

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search