Nomis Data Transformation Service (DTS)

Release 1.0

Joshua Banks
Stelios Boulitsakis Logothetis
Vince Kang
Yanchu Liu
Elias Percy

Mar 21, 2021

CONTENTS:

PREFACE

The following is a usage manual and maintenance guide for our product, the Nomis Data Transformation Service (dubbed DTS).

DTS is a command-line based microservice. It serves to allow users to easily import census data, along with associated metadata, into the Nomis web dissemination tool. It is able to acquire data both from local files (as raw data received directly from the Office for National Statistics) as well as from the Cantabular API, and the service transforms this data into a format that is compatible with the Nomis web dissemination tools. The importing of data into the Nomis system is done via interaction with the Nomis REST API, and the importing of metadata associated with this data is done via Nomis' specialised Metadata API.

This document is partitioned into three sections:

- The **Usage Manual** provides a high-level, non-technical outline of the usage and functionality of the program. Covered here are the pre-requisites for running the program, as well as instructions on how to set the configuration, an explanation of how to run the program and of the different console arguments and modes of operation available, and information on how to run the unit tests included with the code.
- The **Maintenance Guide** provides developer-oriented explanations of the inner-workings of the code. This sections includes many references to and snippets of actual pieces of code from the codebase and contains in-depth instructions on how one may modify the program to suit changing requirements in the future. The program has been written with maintainability and future-proofing as the central design philosophy, so we hope that this section will reflect this.
- The **Documentation** section provides specific insight into all the modules that comprise the program. This includes the various classes, their relationships with other classes, their parameters and methods, and the functions contained within the main pipeline. All classes and methods come with brief descriptions regarding their purpose and functionality. This section is particularly convenient as a reference guide to supplement the previous section (which contains numerous references to the relevant modules detailed here).

Additionally, this document contains two appendices: one containing an exhaustive overview of all arguments that the program has available, and the other containing an complete list of the possible exceptions that the program may raise - which will hopefully be useful for debugging purposes. Finally, the document ends with an index of the program's modules.

CHAPTER

ONE

USAGE MANUAL

1.1 Pre-requisites

The DTS has been written entirely in Python 3.8, so to guarantee a successful run please ensure the version of Python being used is 3.8 or above.

The code makes use of various external libraries which must be installed prior to running the program. These are the following:

- requests (version 2.25.1)
- pyjstat (version 2.2.0)
- chardet (version 4.0.0)

These libraries can all be installed via **pip** in the usual way. Alternatively, we have included a **requirements.txt** file which allows for easy installation of all the required libraries like so:

```
pip install -r requirements.txt
```

This approach is recommended as it ensures the versions of all the libraries correspond precisely with those used during development.

1.1.1 Setting up the Nomis mock APIs

Provided with the source code is an additional folder containing the Nomis mock APIs (with versions compatible for multiple operating systems). The complete setup instructions can be found in the "README.md" within this folder, but for convenience we will also outline them here.

As a prerequisite, these mock APIs require **Microsoft .NET Core**, so please ensure that this is installed on your system prior to testing these.

In the submitted zip file is a folder entitled "mock-apis". The folder of importance is entitled "nomis-api-v0.0.5-metadata-v0.0.2" contained within the "mock-apis" folder.

When in the "nomis-api-v0.0.5-metadata-v0.0.2" folder, the following command-line inputs are needed to compile and run the main Data API:

```
cd fe-api
dotnet build
dotnet run bin/Debug/netcoreapp3.1/fe-api.dll
```

That is, navigate the the "fe-api" folder, input the command "dotnet build" to compile the server, then input "dotnet run bin/Debug/netcoreapp3.1/fe-api.dll" to run the server. Once "dotnet build" has been called once, then only the third command is required to run the server from then on.

Similarly, to compile and run the Metadata API when in the "nomis-api-v0.0.5-metadata-v0.0.2" folder:

```
cd fe-api-metadata
dotnet build
dotnet run bin/Debug/netcoreapp3.1/fe-api-metadata.dll
```

This follows almost the same method as before, but requires navigation to the "fe-api-metadata" folder instead, and a slightly different path for running.

The submitted Data and Metadata APIs have been pre-configured by us to run on ports 5001 and 5005 respectively. If it is necessary to change these ports, then you may do so by locating the "Properties/launchSettings.json" file in each of "fe-api" and "fe-api-metadata", and changing the port directly from there. The servers will have to be restarted for the change to take place. Then the configuration of the DTS must also be suitably changed by updating the "port" values in the "Nomis Connection Information" and "Nomis Metadata Connection Information" sections of the configuration file ("config.json" in the DTS source directory). More detailed instructions on configuring the program are given in the next sections.

UIs for the Mock APIs

The mock APIs come with their own UI hosted by Swagger. These interfaces can be accessed by the following URL:

```
https://localhost:[PORT]/swagger
```

Where [PORT] is substituted with the corresponding port for each API (by default, 5001 for the data API and 5005 for the metadata API). That is,

```
https://localhost:5001/swagger
https://localhost:5005/swagger
```

for the data and metadata APIs respectively.

From there you can perform GET-requests to verify that the imports performed by the DTS were successful.

1.1.2 Connecting to Cantabular

For the program to be able to parse data from online, the Cantabular jsonstat API must be available. As of submission, the URL for the API is:

- https://ftb-api-ext.ons.sensiblecode.io

In addition, the Cantabular API requires authentication. The credentials provided by SCC for our development purposes are:

- Username: durham.project
- Password: extra.carrot.slowly

For the program to run, ensure that in the config.json file under "Cantabular Connection Information", the "address" key has the value "https://ftb-api-ext.ons.sensiblecode.io", and under "Cantabular Credentials", the "username" key has the value "durham.project" and the "password" key has the value "extra.carrot.slowly". This is precisely what is stored in config.json by default, but we are explicitly mentioning it here for the sake of convenience.

1.2 Setting the Configuration

The configuration details of the program are contained within a JSON configuration file. By default, this configuration file will be **config.json**, found in the DTS source directory. The content of this file will dictate the configuration of the program in following runs (this configuration file can be overwritten when a different one is specified in the arguments, but more on arguments in the next section).

By configuration, we are primarily referring to the connection information and credentials for the APIs which the program will have to communicate with.

The config.json file has the following format:

```
{
  "Cantabular Credentials": {
    "username": "string",
    "password": "string",
    "key": null
  },
  "Cantabular Connection Information": {
    "address": "string",
    "port": null
  },
  "Nomis Credentials": {
    "username": "string",
    "password": "string",
    "key": null
  },
  "Nomis Connection Information": {
    "address": "string",
    "port": "string"
  },
  "Nomis Metadata Credentials": {
    "username": "string",
    "password": "string",
    "key": null
  },
  "Nomis Metadata Connection Information": {
    "address": "string",
    "port": "string"
  }
  "Geography Variables": [
    "string",
    "string",
     ...
  ]
}
```

The structure and content of this file have been designed in such a way as to ensure that changing the program's configuration is easy and intuitive.

To change the credentials (username, password, or key) or the connection information (address or port) that the program will use to connect to an API, one only needs to change the corresponding field in the config file. The following constraints apply: the username, password, key, and address values must all be strings. The port can be either a numeric string or an integer, but the numeric value of the port must be within the range of 0 to 49151 inclusive. The key and port fields are both nullable. If the port is null, then it is assumed that the address alone subsumes all the

required connection information, and the program will only use the address provided there to communicate with the API. Moreover, the address must be valid in that it can successfully be resolved into an IP address by the program.

If any of these constraints are disobeyed, the program will detect this early on whilst validating the configuration details, thus raising an error and halting the program prior to commencing connection with any of the APIs.

The "Geography Variables" key contains a list of strings. As of now, this list is as follows:

```
"Geography Variables": [
    "OA",
    "LSOA",
    "MSOA",
    "LA",
    "MERGED_LA",
    "REGION",
    "COUNTRY",
    "Region",
    "Country"
]
```

This list represents all of the variables that come under "geography", for use during the dataset transformations, as these variables are dealt with in a different way compared to the others. If more Geography variables need to be considered, then they can simply be appended to this list. Note that all values here must be strings, otherwise the program's configuration validation will detect this and halt the run.

However, it is important to note that the submitted config file has the geography variables as an **empty list**. This must be the case for testing as the mock APIs are limited in certain functionality, and done fully support the special requirements needed for the geography variables. Thus, the submitted code has the geography variables as an empty list, and this must remain the case for testing purposes.

1.3 Running the Utility

References

Index of Arguments

The program is designed to be run via command line. There is a set of command line arguments for the program, some being mandatory and some being optional. Some arguments are only mandatory if certain other arguments are used. Moreover, there are two *modes* of the program (i.e., data and metadata), and the arguments for each are distinct.

The program is dependent upon the Cantabular API and the Nomis Data API, so if either of these are down or are not correctly configured then the program will raise an appropriate exception and halt. The Nomis Metadata API is also essential when running on metadata mode.

When in the source directory, the program can be run as follows:

```
python main.py {TRANSFORMATION (data | metadata)} {ARGUMENTS}
```

The verb following main.py for selecting the transformation mode (data or metadata) is always mandatory. An exhaustive list of the remaining arguments available for the utility can be found in the *Index of Arguments* section. In particular, of use in both data and metadata mode are the -y, -v, -db, -c, and -l flags. Respectively, these flags provide the functionality of suppressing all (y/n) prompts, enforcing a verbose run of the program (i.e., by logging also to stdout), enforcing "debug" mode (i.e., highly detailed outputs to stdout), overriding the default configuration file path, and overriding the default log file path. The first three flags are simply "toggles" that require no additional input, whereas

the latter two flags each necessitate a string containing the new file paths - the program has methods for ensuring that these exist and/or are valid within the file_reader module (ref, 41).

1.3.1 Data Mode

Running the utility in *data mode* means to transform datasets (either retrieved from the Cantabular API, or from a local file) such that they are in an appropriate format to be transmitted to the Nomis system, and then performing this transmission.

Data mode is selected by following main.py in the command line with the positional argument "data".

```
python main.py data {ARGUMENTS}
```

Two arguments are always required:

- the "-i" flag, which must be followed by a string containing the ID of the Nomis dataset to be created (or updated);
- the "-t" flag, which must be followed by a string containing the title of the aforementioned dataset.

As mentioned, the datasets can be retrieved from Cantabular or from a local file. The approach taken is dependent on the arguments. By default, the dataset will be retrieved from Cantabular via the API - in which case, specified arguments are required for informing the logistics of the query made to the Cantabular API. Specifically, the required arguments when not reading from a file are the "-d" flag, which must be followed by a string containing the name of the (existing) dataset that is to be retrieved from the Cantabular system, and the "-q" flag, which must be followed by a string in the form of a delimited list (e.g., "COUNTRY, SEX") which represent the query variables to be assigned to the Nomis dataset.

Conversely, to obtain the dataset from a locally stored file rather than from Cantabular, the "-f" flag must be used, and the "-d" and "-q" flags shouldn't be used. The "-f" flag must be followed by a string containing the path of the file containing the dataset information, and when used it will automatically toggle "file mode" such that the program will not attempt to query Cantabular.

Note that if the "-f" flag is used in addition to "-d" or "-q", then a prompt will appear declaring that the program detected the use of these contradicting flags and whether the user still wishes to use a file or query from Cantabular instead.

With these flags in mind, an example command line input for initialising a run of the program would be as follows:

```
python main.py data -q "SEX, AGE" -i "SYN456" -t "CENSUS TEST 2" -d
"Usual-Residents"
```

With these arguments, the program will obtain (or at least attempt to obtain) the "Usual-Residents" dataset via the Cantabular API, and this dataset will be transformed in a format suitable for the Nomis system, with the SEX and AGE variables assigned to it.

Alternatively, using the "-f" flag:

```
python main.py data -i "SYN123" -t "CENSUS TEST 1" -f
"examples/cantabular_query_example.json"
```

In this example, instead of querying Cantabular to obtain a dataset, the utility would simply read in "examples/cantabular_query_example.json" and attempt to use this as the dataset instead, otherwise following the same procedures as the previous example. If "examples/cantabular_query_example.json" doesn't exist, the file_reader module (ref, 41) will detect this and raise an exception.

Summary

The utility is set to data mode by using the "data" positional argument immediately following main.py.

The utility can either read from a file (toggled by the "-f" flag, followed by the dataset path) or retrieve a specified dataset from Cantabular (toggled by the "-d" flag, followed by the dataset name, and the "-q" flag, followed by the query variables).

The "-i" and "-t" flags must always be included when using the data mode. These must be followed by the Nomis dataset id and Nomis dataset title, respectively.

The "-y", "-v", "-db", "-c", and "-l" flags can also be included at all times.

1.3.2 Metadata Mode

Running the utility in *metadata mode* entails transforming metadata such that they are in an appropriate format to be transmitted to the Nomis system, and then performing this transmission. Currently, due to uncertainty during development, the utility only supports loading metadata from local files. However, the codebase has been designed such that configuring and updating it in correspondence with future changes can be done with ease. Methods for doing so (including altering the data source such that metadata could be retrieved from Cantabular) are outlined in the following chapter.

In any case, metadata mode is selected by following main.py in the command line with the positional argument "metadata".

```
python main.py metadata {ARGUMENTS}
```

As the utility only supports local files for metadata currently, the "-f" flag is always required. This must be followed by a string containing the path of an existing file containing metadata information. Also, as of now, this file must be in JSON.

Additionally, the "-r" flag is also required. This must be followed by either "O", which indicates that the metadata being read is in the ONS format, or "C", which indicates the metadata is in the Cantabular format. These formats vary so the program must handle them distinctly, thus this flag is required.

An example is as follows:

```
python main.py metadata -f "examples/cantabular_metadata_example.json" -r "C"
```

The above implies that the metadata is being read from a file at "examples/cantabular_metadata_example.json", and this file is in the Cantabular format.

Summary

The utility is set to metadata mode by using the "metadata" positional argument immediately following main.py.

Currently, the utility is only able to read metadata from local files, but the possibility of altering this in the future has been assuredly considered, and information on how this may be done is available in the following chapter.

The "-f" flag is required and must be followed by a path indicating a file containing metadata information, and the "-r" flag, followed by either "O" or "C" is also essential as this will inform the utility on how to handle the raw data.

The "-y", "-v", "-db", "-c", and "-l" flags can also be included at all times.

1.3.3 Input Prompts

Input prompts may appear at multiple points during any given run. As was mentioned earlier, if both the "-f" and "-d" flags are used, the user will be asked whether they wish to continue reading from a file to obtain the data or if they'd rather query Cantabular instead.

If the inputted value for the dataset ID (following the "-i" flag) is the ID of a dataset already existing in the Nomis system, then this will be detected by the Nomis API Connector, and a prompt will be raised asking the user if they wish to overwrite this existing dataset or halt the run instead.

The "-y" flag can be used when starting any run to suppress these prompts, where "y" (i.e. YES) will be selected automatically each time.

1.4 Additional Information

1.4.1 Using the Unit Tests

The included tests for the program are all contained in files prefixed by "test". These files reside in the "tests" folder included with the submission. Unit tests exist for the majority of the program's modules. The tests do not require their respective modules to be in the same directory, so long as the modules are contained in the parent directory. Information on how to run each individual test case is included within comments in each of the test files, but a summary will be provided here.

All tests can be executed from the terminal, and to run all tests contained within a given test file you may execute that file as you would any other python file. That is, using the following format on the command line (using "test_dataset_transformations.py" for the sake of example):

```
python test_dataset_transformations.py
```

This will proceed to execute every test within this file. In this instance, both valid and invalid instances of `DatasetTransformations` will be initialised, and the tests will attempt to use them with both valid and invalid parameters. The tests will assert the expected responses to verify that the class works as it should. The tests for the other modules follow a similar procedure, with variations according to the nuances of each.

Within each test file are multiple distinct tests for separate elements of the modules being tested. For instance, each individual method of `DatasetTransformations` is tested within its own function. These tests can be executed independently using the following syntax:

```
python -m unittest{FILENAME}.{TEST CLASS}.{TEST CASE}
```

To elaborate, the line begins with `python -m unittest`, and this is followed by the filename containing the test class, dot the test class within the file that contains the desired method, dot the method.

For instance, using the "test_dataset_transformations.py" file for the example (the syntax holds for the other test cases as well):

```
python -m unittest test_dataset_transformations.TestDatasetTransformations
.test_dataset_creation
```

It is worth inspecting the test files to gain a better understanding of how this process works, and what specifically is being called.

Note that the test cases for each of the API connectors (that is, the Nomis, Nomis Metadata, and Cantabular API Connectors) require their respective APIs to be reachable. Also, the tests for the main module require all of these servers to be up. Similar dependencies don't exist for the other test cases, where having the modules being tested in a directory prior to the test cases is sufficient for testing.

1.4.2 Type Checking with Mypy

Throughout development we made extensive use of Mypy to encourage and ensure consistent typing. Mypy is a static type-checker for Python, that makes use of Python type hints to check our codebase to make sure no conflicting type definitions or parameters are apparent.

MyPy can be installed with `pip`, using the following command:

```
pip install mypy
```

To use Mypy to statically type-check the codebases, navigate to the folder containing the code and run the command:

```
mypy .
```

That is, "mypy" followed by a full-stop (.). This will cause Mypy to check the entire the typing of codebase, and will raise any issues if it detects any. As of submission, there should be no issues raised as we have dedicated much effort to ensuring typing is consistent throughout.

Note that the test cases are ignored by Mypy, as these often use invalid types purposefully for the sake of testing.

1.4.3 Note on Code Design

Our team has followed the PEP8 standard for the design and format of our code, as this is what we set out to do in our requirements specification. However, we deviated from it with only one exception, in that the maximum line length has been extended from 79 characters to 120 characters. This was done as we felt it helped with the readability of our particular codebase.

CHAPTER

TWO

MAINTENANCE GUIDE

2.1 Logging

References

`main`, 21

The program's logger is initialised within main.py, using Python's built in logging module. Specifically, it is set up by the following section of code:

```
logger = logging.getLogger('DTS-Logger')
logger.setLevel(logging.DEBUG)
logger.propagate = False
formatter = logging.Formatter(
    fmt='%(asctime)s [%(levelname)s; in %(filename)s] %(message)s',
    datefmt='%d/%m/%Y %I:%M:%S %p'
)
file_handler = logging.FileHandler("dts.log")
file_handler.setLevel(logging.INFO)
file_handler.setFormatter(formatter)
logger.addHandler(file_handler)
```

All of the logger's configuration is contained here, so only this code needs to be altered in order to internally configure the system's logging. To change the default log file permanently, one only needs to change the string contained in the `FileHandler` object there. This can be a standalone filename (in which case the file will be made/written to in the same directory as main.py) or a file path, so long as the directory for that path exists. Note that the log file can be changed temporarily during individual runs using the "-l" flag within the command line arguments (refer to Index of Arguments). Moreover, the logger can be made permanently "verbose" by default by changing `logger.propagate` from `False` to `True` (although this too can be set temporarily using the "-v" flag).

By default, the logger established here acts as the "universal" logger throughout the program. Every other module in the program contains the following lines near the top:

```
from logging import getLogger
...
logger = getLogger("DTS-Logger")
```

We felt that the scale of the program lent naturally to having only a single logger, however, we understand that having multiple loggers may be desirable. It is easy to implement this. The code initialising the "DTS-Logger" from above can be used as a template and pasted at the start of any module with a different name and, importantly, a different string (filename) within the `FileHandler`. Alternatively, if separate loggers for multiple modules are desired, one could make use of the type_hints.py module, which is imported by every other module in the program. The logger

can be initialised within the type_hints.py file, and the string within `getLogger` at the start of other modules must simply be changed accordingly.

2.2 Arguments

References

`args_manager`, 24
`arguments`, 24

All code pertaining to the *arguments* used in the program is contained in two modules: args_manager.py and arguments.py. Within the former, the ArgsManager class handles the parsing of the arguments from the terminal, primarily making use of the argparse library module. Within the latter, the Arguments class stores the arguments and contains a method for validating them.

Adding arguments to the program is simple. Within the ArgsManager, the constructor is home to an argparse parser object, and all the arguments available within the program are added to the parser sequentially there. For instance,

```
self.parser.add_argument(
            "-f",
            "--filename",
            action="store",
            help="Read data from a file instead of querying Cantabular.",
            dest="filename",
            type=str,
            default=None
)
```

The code above presents an example of an "-f" flag being added to the available arguments of the program. To append an additional argument, one simply has to use that template, changing the values and types as fit. As long as the values don't conflict with existing arguments, this code can simply be inserted at the bottom of the constructor.

Then, to ensure this new argument is stored within the Arguments class, one simply has to insert a new class variable into the constructor of Arguments. This is done in the same way that the other class variables are initialised. For instance,

```
self.filename = arguments.filename
```

The above demonstrates how an argument is established as a part of the Arguments class. Notice that the attribute of arguments used to retrieve the argument is the name stored in the "dest" attribute of the parser. After this, steps for validating this argument may be added to the validate() method of Arguments.

Removing an argument is similarly intuitive, but substantially more dangerous. To do it, simply remove the parser method from the ArgsManager constructor, and then remove the corresponding class variable from the constructor of the Arguments class. Moreover, one must also remove any methods in the validate() method of Arguments that are contingent on this variable. Note that this is dangerous because other parts of the program may rely on the argument, so only delete an argument if the codebase has been correspondingly altered such that no other component uses the argument to be deleted.

2.3 Configuration

References

config_constants, 26
config_manager, 26
connection_info, 27
credentials, 27
configuration, 28

The configuration of the program is handled within four modules: config_manager.py, connection_info.py, credentials.py, and configuration.py. Additionally, there is a file config_constants.py that contains a path to the default configuration file, and a python dict representation of the default configuration file. Moreover, the user-customisable configuration information is contained within config.json.

As shown previously, config.json has the following structure:

```
{
  "API Credentials": {
    "username": "string",
    "password": "string",
    "key": null
  },
  "API Connection Information": {
    "address": "string",
    "port": "string"
  },

  ...

 "Geography Variables" : [
   "string": "string",
    ...
 ]
}
```

The values here are fully customisable so as to correspond with how the system is set up. Before running the program, ensure that the values in the config file correctly correspond with what they should be; i.e., the port and address for each API should be set to where that API is served.

A config file separate from the default one (config.json) may be selected at run time, using the "-c" flag. However, this file must exist and be in the format as displayed above (and must be a JSON), else the program will detect it as being an invalid config file and either attempt to use the default one instead or halt the program.

The content of the configuration file will be collected by the ConfigManager class, and an instance of Configuration is constructed based on this content via the ConfigManager's decode_configuration() method. This is done by constructing instances of Credentials and ConnectionInfo for each of the APIs included in the configuration file, validating these within their respective classes, and then storing them in an instance of Configuration.

The decode_configuration() function contains the following list:

```
api_config_details = [
            # FORMAT: ("API Name", "Credentials key", "Connection info key"),
            (
                "cantabular",
```

```
                "Cantabular Credentials",
                "Cantabular Connection Information"
            ),
            (
                "nomis",
                "Nomis Credentials",
                "Nomis Connection Information"
            ),
            (
                "nomis_metadata",
                "Nomis Metadata Credentials",
                "Nomis Metadata Connection Information"
            )
]
```

That is, a list containing tuples in the format specified by the comment preceding the content of the list. The content of this list dictates how the method goes on to "decode" the configuration based on the config file. Notice that the "keys" here correspond with the JSON config file shown earlier.

If the program ever requires an additional API, one can add one simply by adding to the configuration JSON two new keys, one containing the credentials for this new API and one containing the connection information. Then, these new keys must be added to this list in the decode_configuration() method, in the way that the others had been added in the example above. Once this is done, the new configuration details will now be entirely accessible to the program. Consult the documentation for the Configuration class for more information on how this is done. In short, the credentials and the address of each API can be easily obtained by an instance of Configuration by reference to the name assigned to it (i.e., the first item in its respective tuple).

Regarding the "Geography Variables" key, if more such "geography" variables need to be added, then simply appending the names of such to this list will be sufficient, as the program will automatically handle all values in this list appropriately as geography variables. However, if an entirely new section/type of variables is needed, then a new list would need to be added to the config file. This can be done trivially, but the `DatasetTransformations` class will likely also have to be altered. More information on how to so can be found in the section dedicated to this (ref, here), but note that this class has been designed in such a way as to allow for easy configuration.

2.4 Handling Files (The FileReader Class)

References

`file_reader`, 41
`dataset_file_reader`, 31

The purpose of the `FileReader` class is to allow for more simplicity with regards to the handling of files throughout the program. It is used for:

- Reading and validating both default and custom config files, and writing out the default one (JSON) if necessary;
- Reading and validating custom log files;
- Reading in and validating data/metadata files;
- Writing out requests, request bodies, and responses in a text file for the API Connectors.

It is convenient in that it allows for a reduction in repeated code, and for validating the existence of a file before attempting to handle it.

Currently, it has a method for loading in json files (i.e., load_json()). This is as follows:

```
def load_json(self) -> Union[dict, str]:
        enc = chardet.detect(open(self.file, 'rb').read())['encoding']
        with open(self.file, 'r', encoding=enc) as json_file:
                data = json.load(json_file)
        return data
```

Note that the file is stored as a class attribute (`self.file`). Of importance is the fact that this class is inherited by the `DatasetFileReader` class. The JSON file-format is currently used for storing data and metadata files locally. However, this is subject to change; if the file-format does change, an additional method needs to be added to handle that format. This is generally fairly simple to do, and a good idea of how to do so can be attained by looking at the syntax of the existing method. The usage of load_json() within the query() method of `DatasetFileReader` would also need to be changed correspondingly.

Note that the load_json() method shouldn't be *replaced*, as it is used for other purposes throughout the program as outlined above (e.g. for reading the config file).

2.5 Data Source

References

`data_source`, 30
`dataset_file_reader`, 31
`cantabular_api_connector`, 30

The `DataSource` class is fundamental to obtaining the data to be transformed. It is inherited by the `CantabularApiConnector` and `DatasetFileReader` classes. The class is composed of an abstract method, query(), which is used for the initial retrieval of the data, and the static method load_jsonstat(), which takes the data retrieved within query() and converts it into a pyjstat Dataset object for use throughout the transformations. Note that the version of query() in the `CantabularApiConnector` class and the `DatasetFileReader` class both call load_jsonstat().

We understand that both the format and source of data that the program will need to deal with is subject to change after the product handover, as it has been throughout development. The following will outline how the `DataSource` class and its child-classes can be altered to address such changes.

The query() method of `DataSource` was deliberately made abstract, so it can be filled in accordance with where the data is retrieved from. For instance, the query() method of `CantabularApiConnector` entails querying the Cantabular API to download a dataset stored on their system then using this dataset as a parameter for the `DataSource` load_jsonstat() method. This requires the "-d" flag. The `DatasetFileReader`'s query method, on the other hand, makes use of the `FileReader` class to read in the JSON file (after confirming its existence), then calling the load_jsonstat() method in the same way. This requires the "-f" flag.

The content of each query() method is rather lightweight and easily configurable. If the file-format of locally stored data files changes, then refer to the previous section regarding how to alter the `FileReader` class to account for this - the query() method of `DatasetFileReader` would simply only need to call the new method instead of the load_json() method as it currently does.

The process of making a request to Cantabular is contained entirely within the query() method of `CantabularApiConnector`. If this needs to be changed (for instance, if the type of request that is needed to be made, or the content of the request body changes, etc.), then only this segment of code needs to be altered. The endpoint to which the request will be made is established in the constructor of `CantabularApiConnector`, in the following line:

```
self.query_url = self.client + '/v8/query-json-stat/%s?%s' % (dataset,
'&'.join([f'v=v' for v in variables]))
```

If this endpoint changes, then one only needs to change the string that follows `self.client` in the line above, assuming the configuration of the program has also been changed in accordance.

2.6 Dataset Transformations

References

`dataset_transformations`, 31

The methods and operations for transforming data and metadata into a format ready for transmission to the Nomis system are all contained within the `DatasetTransformations` class. In general, only these methods (potentially with some minor tweaks to the Nomis API Connectors as well) will need to be edited if the format or content of the data that the Nomis system will accept changes.

In addition to some validation methods, the class begins with a static method named `dataset_creation()`, which takes two strings as parameters (representing *dataset ID* and *dataset name*, in that order). This method is used in data mode, and it serves to establish the dataset in the correct format that will promptly be transmitted to the Nomis database via the connector. This, in a sense, begins the lifecycle of the utility on data mode, after all arguments and configuration details have been validated. If the format of dataset that Nomis accepts changes, then only this method needs editing. Currently, the dataset is initialised as a Python dict with a clear set of key-value pairs. If any existing key or value is altered, ensure the `validate_ds()` method of the Nomis API connector won't now block the dataset (for instance, if a value that was initially *None* is changed to some string, ensure the validation method now allows for this value to be a string).

The methods that follow `dataset_creation()`, up until the first (and only) method dedicated to metadata, all sequentially correspond with the data transformation life cycle. Included are methods for:

- Variable Creation
- Type Creation
- Category Creation
- Dataset Assignment
- Observations

All of these methods require the pyjstat dataset either retrieved from Cantabular or a local file to process these transformations. In main.py, when any of the aforementioned methods are used, they are followed by a corresponding method from the `NomisApiConnector` class for importing them onto the Nomis system. Generally speaking, if any of these objects (i.e., variables, categories, observations, and so on) require a different format from Nomis, then the transformation method within this class that corresponds with that object needs to be changed. Moreover, we ensured that making such changes wouldn't be too difficult be limiting the amount of the procedure that we abstracted in this methods. For instance, the construction of the dimensions (as an arbitrary example) is done in a very clear manner, as in the Python dictionary is clearly laid out and defined. Thus, altering the way these objects are initialised, their content, or their structure is to be as simple a task as possible.

To exemplify this notion, the following is taken directly from the codebase, within the `dataset_creation()` method, at the point where the Python dictionary object representing the dataset is defined:

```
ds = {
    "id": dataset_id,
    "title": dataset_title,
```

```
    "contactId": "Census",
    "isAdditive": True,
    "isFlagged": False,
    "derivedFrom": None,
    "restrictedAccess": False,
    "minimumRound": 0,
    "online": True
}
```

Here, the `dataset_id` and `dataset_title` are method parameters, and this object is promptly returned by the method. If the format needs to be changed, then it likely will only need to be changed here.

The method for transforming metadata into a Nomis-acceptable format is also contained within this class. This method is the last method of the class, structurally, and is named `variable_metadata_request()`. Of importance is that this method is a *static method*, meaning an instance of `DatasetTransformations` is not required to be initialised when handling metadata, and in fact it isn't in the current version of the program.

This method contains within it the entire process of transforming the metadata into the format ready for the Nomis system. It takes a its parameter a list of `UuidMetadata` namedtuples, which each contain a UUID and a description for the metadata with that UUID. This list is collected within one of the metadata functions in main.py. If the metadata format requires change at some point in the future, then it's likely that this method alone will need to be altered. In it, the construction of metadata objects is made clear in the same way as above: a single Python dictionary definition is visible within a list comprehension; changing the content or format of this will universally change the format of metadata that Nomis receives from the system. The following is a snipped of this list comprehension:

```
requests = [
    {
        "id": None,
        "belongsTo": id_md.uuid,
        "description": None,
        "created": None,
        "validFrom": None,
        "validTo": None,
        "include": None,
        "meta": [
            {
                "role": "note",
                "properties": [
                    {
                        "prefix": "dc",
                        "property": "description",
                        "value": id_md.metadata["description"]
                    }
                ]
            }
        ]
    }
    for id_md in uuids_metadata]
```

Here, `uuid_metadata` is the previously mentioned parameter to this method. Changing the types or values of the key-value pairs in this dictionary in response to any shifting requirements is straightforward.

2.7 Nomis API Connectors (Data and Metadata)

References

api_connector, 40
nomis_api_connector, 33
nomis_metadata_api_connector, 38

All of the communication with the two Nomis APIs is done via the two Nomis API Connector classes; that is, `NomisApiConnector` and `NomisMetadataApiConnector`.

Both of the Nomis API connectors inherit the `ApiConnector` class. This class contains methods for establishing a requests Session with the Nomis client and a method that uses the `FileReader` class to write out request and response details to a file.

Within the `NomisApiConnector` and `NomisMetadataApiConnectors`, distinct methods exist for every request that the program needs to make to the respective APIs. In particular, the ordering of these methods is deliberately sequential in relation with the order that they are used by the program. These methods all follow a similar template: beginning with validation for the paramters, then followed by a `try-except` statement in which the request is made, and then the response is handled according to its status code.

The following is an example of one such method - in this case, it is the method for creating datasets via the `NomisApiConnector` class:

```
def create_dataset(self, id: str, ds: NomisDataset) -> bool:

    # Type/value checking
    self.validate_ds(ds, id)
    self.validate_id(id)

    # Make the request: Update/create a dataset.
    headers = {'Content-type': 'application/json', 'Accept': 'application/json'}
    try:
        res = self.session.put(
            f'{self.client}/Datasets/{id}',
            data=json.dumps(ds),
            headers=headers,
            verify=False
        )
    except Exception as e:
        raise requests.ConnectionError(f"Unable to connect to client. ({e})")

    self.save_request("create_dataset()", res)

    # Handle response
    if res.status_code == 200:
        logger.info("Dataset created successfully.")
        return True
    elif res.status_code == 400:
        raise requests.HTTPError("Bad input parameter.")
    elif res.status_code == 404:
        logger.info(res.text)
        raise requests.HTTPError(f"Dataset (id: '{id}') already exists.")
    else:
```

```
        raise Exception(f"Unexpected response with status code {res.status_code}.")
```

As you can see, the method is loosely partitioned into three sections: validation, request, response. In addition, there is a call to the `save_request()` method; this is the method mentioned earlier that servers to save request the request and response to a specific file within a folder that is created along with the class instance.

The structure of the above method is very consistent throughout the other methods within each Nomis API connector class, and we have deliberately formatted them in a clear, segmented way to allow for easier configuration. If the interaction with the Nomis APIs needs to change, then only these two classes should need to be configured, which would have no impact on the other components of the program. For instance, if a new HTTP method is required, or an additional endpoint is to be involved, then one of the pre-existing methods can be duplicated and appropriately altered.

2.8 Main

References

`main`, 21

The `main` module takes a functional design, with the various parts of the program's pipeline being handled within distinct functions, which all make calls to the various modules of the program. In particular, there are two "main" functions - main_data() and main_metadata(). These centralise each mode of the utility and, on observation, provide a high-level overview of the processes.

The first few functions of main.py, which come under the "Initialisation" section, are for collecting the arguments and collecting the configuration of the program. These functions communicate with the `ArgsManager` and `ConfigManager` respectively. Thus, if changes to those two classes that involve additional attributes or methods are made, and these need to be accessed directly by the program, then these functions are to be consulted.

The next section of main.py contains the functions for "data mode", which are all accessed by the previously mentioned `main_data()` function (which resides at the bottom of the file, in a section for the two main functions). The code has been divided such that each part of the process is contained within its own function. Moreover, even though only one instance of `DatasetTransformations` is created, all of its distinct methods are called within independent functions, along with corresponding methods from `NomisApiConnector` (note also that only one instance of this is used).

The purpose of this separation is to allow for easier configuration. For instance, if any of the dataset transformation methods change, then ideally only one function within main.py will need to also be considered.

The `main_data()` function itself is as follows:

```
def data_main() -> None:
    with NomisApiConnector(
            config.get_credentials('nomis'),
            config.get_client('nomis')
    ) as connector:
        exists = check_dataset_exists(connector)
        dataset_transformations(connector, exists, retrieve_data())
    logger.info(f"SUCCESS: A dataset with the ID {args.dataset_id} has been "
                f"{'UPDATED' if exists else 'CREATED'} successfully.")
```

Note that the only functions called directly by this main function are `check_dataset_exists()` and `dataset_transformations()`. The divided dataset transformation functions are all called within the umbrella `dataset_transformations()` function. This main method was made as visually lightweight as possible as we intend for it to double as a high level overview of the procedures.

The `main_metadata()` function is as follows:

```
def metadata_main() -> None:

    with FileReader(args.filename) as fr:
        file_data = fr.load_json()

    if args.metadata_format.lower() == 'c':
        uuids_metadata = cantabular_metadata(file_data)
    elif args.metadata_format.lower() == 'o':
        uuids_metadata = ons_metadata(file_data)
    else:
        raise ValueError("Unrecognised metadata format.")

    if len(uuids_metadata) > 0:
        variable_metadata_requests = DatasetTransformations.variable_metadata_request(
            uuids_metadata
        )

        with NomisMetadataApiConnector(
                config.get_credentials('nomis_metadata'),
                config.get_client('nomis_metadata')
        ) as metadata_connector:
            uuids = metadata_connector.add_new_metadata(
                variable_metadata_requests,
                return_uuids=True
            )
        logger.info(f"Metadata handled successfully. {uuids}")

    else:
        logger.info("No metadata appended.")
```

It begins by using the `FileReader` class to read in the file containing metadata. If, in the future, new methods for obtaining the metadata other than from a file become available, then this section of the function can be tweaked to include a conditional statement as well as a call to an additional module for handling the retrieval of metadata in this new way.

The two primary functions called within `maon_metadata()` are `cantabular_metadata()` and `ons_metadata()` - either one or the other depending on the arguments. These functions can be found in the "Metadata Functions" section of main.py. In summary, each of these functions handles metadata according to the Cantabular format and the Nomis format, as these two formats were available to us during development. The functions compile UUIDs and metadata descriptions corresponding with each based on the format of metadata they have been constructed to handle. The return type is a `namedtuple` which is then used in the `variable_metadata_request()` method of the `DatasetTransformations` class in order to construct metadata objects suitable for transmission via the Nomis metadata API (using the `NomisMetadataApiConnector` module).

If more formats of metadata arise, then a new function will need to be constructed to handle this. The operations within this new function will of course be dependent on the new format of metadata it must handle, but it is important that the function returns an instance of the `UuidMetadata` namedtuple, containing the UUID and the metadata description of each of the new pieces of metadata extracted from this new format. This is because the Metadata transformation method has been written to accept these namedtuples and generate the Nomis metadata from them. Furthermore, the specifics of the metadata format to be transmitted to Nomis can be configured within the aforementioned `variable_metadata_request()` - so if the accepted format of metadata on the Nomis end changes, this method should be consulted.

2.9 Updating the Test Cases

If any of the modules in the program are altered, it's likely that their associated test case(s) will need to be altered correspondingly.

The test cases all follow a similar pattern, with distinctions based on the various nuances of each module. Generally, each test file contains one class representing the unittest for that module, and within that class the various methods of the module are tested in separate functions, or the same methods are tested using varying arguments or parameters. These classes inherit `unittest.Testcase` from Python's unittest library.

Usually, each individual test function beings by testing invalid approaches to the associated method and asserting that the correct response is retrieved (e.g., if an invalid argument is passed, a `TypeError` or `ValueError` is raised in accordance). Then, valid approaches are tested, and it is asserted that the expected values are returned (or, in some cases, `stdout` receives the expected messages, or the Nomis database now contains a specified object). When testing the Nomis API connectors, most methods entail actually posting or appending items via the APIs, and thus these methods end with a delete request to undo the requests made. This allows for repetition of the tests without false positives (or negatives). It is recommended that the mock or local version of these APIs are used when testing as it is potentially dangerous to do so on the live server.

Extending the test cases is fairly straightforward. For instance, if a new method is added to the Nomis API connector, then a new test can be added to its associated test case by following a similar format to the other test methods.

CHAPTER

THREE

DOCUMENTATION

3.1 Main

3.1.1 main.py

main.**collect_arguments**()
Use the Args Manager to establish the arguments for given run.

Return type *Arguments*

Returns An instance of *Arguments* containing the argument data established for a given run.

main.**collect_configuration**(*arguments*)
Use the Config Manager to establish the configuration for a given run.

Parameters **arguments** (*Arguments*) – An instance of *Arguments* containing the argument data established for a given run.

Return type *Configuration*

Returns An instance of *Configuration* containing the established configuration details for a given run.

main.**check_dataset_exists**(*connector*)
Check whether the dataset already exists in the Nomis database, handle appropriately.

Parameters **connector** (*NomisApiConnector*) – An open, initialised instance of *NomisApiConnector*.

Return type bool

Returns A bool indicating if the dataset does exist (*True*) or it doesn't exist (*False*).

main.**retrieve_data**()
Query the Cantabular API to retrieve a jsonstat table for use in dataset construction and transformation.

Return type Tuple[Dataset, List[str]]

Returns A tuple containing a valid pyjstat dataset, retrieved from cantabular or a file, and a list of query variables, retrieved from the arguments or from a file.

main.**check_dataset_dimensions**(*connector*, *dimensions*)
Obtain a list of all variables marked for posting that haven't already been assigned to the dataset.

Parameters **connector** (*NomisApiConnector*) – An open, initialised instance of *NomisApiConnector*.

Return type bool

Returns A list of variables (from the arguments) that have not yet been assigned to the dataset.

main.**get_type_ids**(*type_requests*)
Obtain a list of all unique type ID.

Parameters **type_requests** (List[dict]) – A list of type requests.

Return type List[str]

Returns A list of type ids.

main.**create_dataset**(*connector*, *transformations*)
Initialise a dataset using the jsonstat table either read in or retrieved from Cantabular.

Parameters

- **connector** (*NomisApiConnector*) – An open, initialised instance of *NomisApiConnector*.
- **transformations** (*DatasetTransformations*) – An initialised instance of *DatasetTransformations* with a valid table attribute.

Return type None

main.**handle_variables**(*connector*, *transformations*, *variables*)
Handle variable transmission/manipulation.

Parameters

- **connector** (*NomisApiConnector*) – An open, initialised instance of *NomisApiConnector*.
- **transformations** (*DatasetTransformations*) – An initialised instance of *DatasetTransformations* with a valid table attribute.
- **variables** (List[str]) – A list of variables to be assigned to the dataset.

Return type None

main.**handle_dimensions**(*connector*, *transformations*, *key*)
Assign dimensions to the dataset.

Parameters

- **connector** (*NomisApiConnector*) – An open, initialised instance of *NomisApiConnector*.
- **transformations** (*DatasetTransformations*) – An initialised instance of *DatasetTransformations* with a valid table attribute.

Return type None

main.**handle_observations**(*connector*, *transformations*)
Append/overwrite observations to the dataset.

Parameters

- **connector** (*NomisApiConnector*) – An open, initialised instance of *NomisApiConnector*.
- **transformations** (*DatasetTransformations*) – An initialised instance of *DatasetTransformations* with a valid table attribute.

Return type None

main.**dataset_transformations**(*connector*, *exists*, *data*)
Function containing the dataset transformation operations.

Parameters

- **connector** (*NomisApiConnector*) – An open, initialised instance of *NomisApiConnector*.
- **exists** (bool) – A bool indicating whether or not the dataset currently exists; if *True*, then the function will handle for updating an existing dataset. Conversely, the function will create a new dataset if exists is *False*.
- **data** (Tuple[Dataset, List[str]]) – A tuple containing the required data. That is, a pyjstat dataset corresponding with the query made to cantabular, and the list of variables to be assigned to the dataset.

Return type None

main.**cantabular_metadata**(*file_data*)

Function for handling metadata in the Cantabular format.

Parameters **file_data** (dict) – A dictionary containing a Python dict representation of the inputted metadata file.

Return type List[*UuidMetadata*]

Returns A list of namedtuples containing a UUID (str) and its associated metadata (dict).

main.**ons_metadata**(*file_data*)

Function for handling metadata in the ONS format.

Parameters **file_data** (dict) – A dictionary containing a Python dict representation of the inputted metadata file.

Return type List[*UuidMetadata*]

Returns A list of namedtuples containing a UUID (str) and its associated metadata (dict).

main.**data_main**()

Main function for handling datasets.

Return type None

main.**metadata_main**()

Main function for handling metadata.

Return type None

3.2 Arguments and the Args Manager

3.2.1 args_manager.py

class args_manager.**ArgsManager**
Bases: object

Class for handling the input arguments, utilising the argparse library module. Allows for command line arguments in the following formats:

For handling *data*: main.py data -f {FILENAME (optional)} -q {QUERY in quotes} -i {ID} -t {TITLE} -y (for yes to all prompts) -v (for verbose)

For handling *metadata*: main.py metadata -f {FILENAME} -r {METADATA FORMAT}

Variables **parser** (*ArgumentParser.*) – An argparse *ArgumentParser* object for collecting arguments from the terminal.

decode_arguments()
Method for decoding the arguments collected by the parser into an instance of *Arguments*, and then calling the *Arguments* validate() method.

Return type *Arguments*

Returns A validated instance of *Arguments* corresponding with the terminal inputs.

3.2.2 arguments.py

class arguments.**Arguments**(*arguments*)
Bases: object

Container class for the program's arguments; includes method for validating the arguments.

Parameters **arguments** (Namespace) – Parsed argparse object containing the arguments for the program.

Variables

- **transformation** (*str*) – String indicating whether the metadata or normal (i.e. data) program pipeline is used.
- **metadata** (*bool*) – A bool that will be resolved to *True* if metadata mode is toggled. Otherwise, it will be *False*.
- **suppress_prompts** (*bool*) – Toggle for suppressing prompts.
- **verbose** (*bool*) – Toggle for a verbose out during runtime.
- **filename** (*Optional[str]*) – Location of a file to read from instead of querying Cantabular.
- **query_variables** (*Optional[List[str]]*) – Parameter for querying Cantabular.
- **dataset_id** (*Optional[str]*) – Parameter for querying Cantabular.
- **dataset_title** (*Optional[str]*) – Parameter for querying Cantabular.
- **query_dataset** (*Optional[str]*) – Parameter for querying Cantabular.
- **log_file** (*Optional[str]*) – For overriding the default location of the log file.
- **config_file** (*Optional[str]*) – For overriding the default location of the config file.

validate()

Method for validating the arguments, and raising an exception in the case of anything invalid.

Return type bool

Returns Returns *True* upon successful validation; otherwise, an exception will have been raised.

Raises

- **ValueError** – If any included argument contains an empty string, or any required argument is excluded.
- **FileNotFoundError** – If the inputs for *filename* or *config_file* aren't paths to existing files.
- **IOError** – If the arguments for *filename* or *config_file* aren't suffixed by '.json', or if *log_file* suffix isn't '.log'.

3.3 Configuration and the Config Manager

3.3.1 config_constants.py

This module simply contains *constants* for use by the *ConfigManager*. This includes the default config file encoded as a string.

3.3.2 config_manager.py

class config_manager.**ConfigManager**(*args=None*)
Bases: object

Class for handling the program configuration, including reading from default or customised file paths, writing default files, and decoding JSON configuration into relevant classes

Parameters **args** (Optional[*Arguments*]) – An instance of the arguments manager, which may contain an alternate config file

Variables

- **config** (*dict*) – A python dict containing the contents of the config file that the program will use.
- **default** (*dict*) – A python dict containing the default configuration information, in case the inputted or enforced one contains errors.

write_default()
Write a default config file if one doesn't exist, utilising the *FileReader* class.

Return type None

decode_credentials(*key*)
Create an instance of Credentials based on the content in the config file for the 'key' parameter.

Return type *Credentials*

Returns A valid instance of *Credentials*.

decode_connection_info(*key*)
Create an instance of ConnectionInfo based on the content the config file for the 'key' parameter.

Parameters **key** (str) – This corresponds to the name of the attribute in the JSON config file that contains the connection information for a certain API - e.g. 'Nomis Connection Information'.

Return type *ConnectionInfo*

Returns A valid instance of *ConnectionInfo*.

decode_geography_variables(*key*)
Create an instance of the geography variables based on the content of the config file for the 'key' parameter

Parameters **key** (str) – The 'key' corresponding to the Geography variables in the config file.

Return type List[str]

Returns A list containing the geography variables.

decode_configuration()
Create an instance of Configuration by combining an instances of Credentials and ConnectionInfo for each of the APIs that the program will communicate with.

Return type *Configuration*

Returns A validated instance of *Configuration*.

3.3.3 connection_info.py

class connection_info.**ConnectionInfo**(*address*, *port*)
Bases: object

Class for containing and validating the connection information (address and port) for each API.

Parameters

- **address** (str) – A string of a valid address, either a URL or IP address, for connecting to the API.
- **port** (Union[str, int, None]) – A string or an integer representing the port the API will be served on.

Variables

- **address** (*str*) – Initial value: *address*.
- **port** (*Union[str, int]*) – Initial value: *port*.

validate()
Method for validating the ConnectionInfo attributes. For this class, all attributes are mandatory, and so need to be successfully validated. In this case, the address must be a valid IP address, or must be able to be successfully resolved into one (i.e., it can be a valid URL), and port must be a valid numerical string, or an integer, and its numerical value must be within an acceptable range.

Raises

- **TypeError** – If the *address* is not a valid string, or the *port* is not a valid numeric string or integer.
- **ValueError** – If the *address* is an empty string, or if the numeric value of the *port* is not within an acceptable range (i.e., between 0 and 49151, inclusive).

Return type bool

Returns *True* if the validation is successful; otherwise, an exception will have been raised.

3.3.4 credentials.py

class credentials.**Credentials**(*username*, *password*, *key=None*)
Bases: object

Class for containing and validating API credentials.

Parameters

- **username** (str) – A string containing a valid username for authenticating the associated API.
- **password** (str) – A string containing a valid password for authenticating the associated API.
- **key** (Optional[str]) – A string containing a valid key for accessing the associated API.

Variables

- **username** (*str*) – Initial value: *username*.

- **password** (*str*) – Initial value: *password*.
- **key** (*Optional[str]*) – Initial value: *key*.

validate()
Method for validating the Credentials class attributes. For this class, all attributes other than the *key* are mandatory, and so need to be successfully validated (in this case, they must all be valid, nonempty strings). Validation will still be successful if no key is passed.

Raises

- **TypeError** – If any of the *username*, *password*, or *key* (if not *None*) are not valid strings.
- **ValueError** – If the *username*, *password*, or *key* (if not *None*) are empty strings.

Return type bool

Returns *True* if validation is successful, otherwise an exception will have been raised.

3.3.5 configuration.py

class configuration.**Configuration**(*config*, *var=None*)
Bases: object

Class to hold instances of Credentials and ConnectionInfo for all of the APIs that the program communicates with, in addition to the geography variables that must be considered when the program handles variables. This class is used frequently throughout the program, as it is the primary reference point as regards the program's settings for any given run.

Parameters

- **config** (Dict[str, *CredentialsConninfo*]) – A list of `namedtuple`s containing instances of ConnectionInfo and Credentials for all APIs.
- **var** (Optional[Dict[str, List[str]]]) – Dictionary of special variables that must be acknowledged by the program.

Variables

- **config** (*Dict[str, CredentialsConninfo, List[str]]*) – Initial value: config.
- **var** (*Optional[Dict[List[str]]*) – Initial value: var.

get_credentials(*api*)
Method for returning the credentials for a given API in the form of a tuple. Convenient for more swift API authentication.

Parameters **api** (str) – String representing the API credentials to receive: nomis, nomis_metadata, or cantabular.

Raises **NameError** – If the API name passed is not recognised by this instance.

Return type Tuple[str, str]

Returns A tuple containing the username and password for the request API, respectively.

get_client(*api*)
Method for returning the concatenation of the address and port of an API. Useful for more swift API connection.

Parameters **api** (str) – String representing the api to receive the client for: nomis, nomis_metadata, or cantabular.

Raises **NameError** – If the API name passed is not recognised by this instance.

Return type str

Returns A string representing a concatenation of the address and the port, separated by a colon.

get_geography()
Method for returning a list of the geography variables that the program must consider.

Raises **KeyError** – If the geography variables haven't been defined in the configuration.

Return type List[str]

Returns The geography variables, as strings in a list.

3.4 Data Source and Transformations

3.4.1 data_source.py

class data_source.**DataSource**
Bases: object

This class represents the retrieval of data and returning it in the form of a pyjstat Dataset. The query() method has consciously been made abstract to allow for simpler configuration should another class inherit this class and is required to retrieve the data from a new source. Currently, this class is inherited by *CantabularApiConnector* and *DatasetFileReader*, with their query() methods customised to allow for retrieval of data via the Cantabular API and via local files, respectively.

abstract query()
Abstract method for either querying Cantabular directly, or generating a Cantabular-style jsonstat dataframe using the information from a local file; will be altered in accordingly by classes who inherit this.

Return type Dataset

Returns A pyjstat Dataframe containing the query information (obtained via the load_jsonstat() method).

static load_jsonstat(*data*)
Static method for taking a JSONstat string and returning a valid/verified pyjstat dataframe.

Return type Dataset

Returns A pyjstat Dataframe containing the query information.

3.4.2 cantabular_api_connector.py

class cantabular_api_connector.**CantabularApiConnector**(*dataset*, *variables*, *credentials*, *address*, *port=None*)
Bases: *api_connector.ApiConnector*, *data_source.DataSource*

Class for communicating directly with the Cantabular API, in order to retrieve a jsonstat dataset based on specific queries. Currently only configured to work with data (i.e., not metadata).

Parameters

- **dataset** (str) – Name/ID of a dataset to retrieve from the Cantabular system.
- **variables** (list) – A list containing valid variables.

Variables **query_url** (*str*) – URL with endpoints derived from params dataset and variables.

query()

Method for making a query to the Cantabular API using the argument variables. This is the Cantabular API connector version of the query() abstract method, inherited from the *DataSource* class.

Raises **requests.HTTPError** – Raised in the case of a network partition or invalid query to the Cantabular API.

Return type Dataset

Returns A cantabular table in the form of a jsonstat dataframe.

3.4.3 dataset_file_reader.py

class dataset_file_reader.**DatasetFileReader**(*file*)
Bases: *file_reader.FileReader*, *data_source.DataSource*

Class for handling datasets read from files locally rather than queried directly from Cantabular. Inherits from *DataSource* with its query() method configured for the retrieval of data from local files, and the *FileReader* class is also inherited to assist with this retrieval (and verification).

query()
The file equivalent of querying cantabular; i.e., ensuring the inputted file exists and converting it into a usable pyjstat Dataset (table).

Raises **FileNotFoundError** – If the input file path can't be located.

Return type Dataset

Returns A cantabular table in the form of a jsonstat dataframe.

3.4.4 dataset_transformations.py

class dataset_transformations.**DatasetTransformations**(*geography_flag*, *table=None*, *table_geography=None*)
Bases: object

Class containing methods for transforming the data retrieved from Cantabular such that it is ready for transmission to the Nomis system.

Parameters **table** (Optional[Dataset]) – A pyjstat dataframe containing the data to be used/transformed.

Variables **table** (*Dataset*) – Initial value: table.

validate_table()
Method for validating the table, ensuring it is of the correct type and contains sufficient keys.

Raises

- **TypeError** – If the dataset is not of the correct type, i.e., a pyjstat Dataset.
- **LookupError** – If the table is devoid of a "dimension" key, which is required for the transformations.

static dataset_creation(*dataset_id*, *dataset_title*)
Method for initialising a Nomis-style dataset using an inputted ID and title.

Parameters

- **dataset_id** (str) – A valid string representing the ID of the dataset to be created.
- **dataset_title** (str) – A valid string representing the title of the dataset to be created.

Raises

- **TypeError** – If *dataset_id* or *dataset_title* are not valid strings (str types).
- **ValueError** – If *dataset_id* or *dataset_title* have a length of 0 (i.e., are empty).

Return type Dict[str, object]

Returns If no exception is raised, an initialised dataset as a Python dict with valid keys and values.

variable_creation()
Method for converting the variable data from the pyjstat dataframe into valid variables for communication with the Nomis API.

Return type List[Dict[str, object]]

Returns A list of variables.

type_creation()
Method for creating a valid variable type using the pyjstat dataframe for communication with the Nomis API

Raises **KeyError** – If the dataframe doesn't contain any dimension information

Return type List[Dict[str, object]]

Returns A list of types (should usually be a single dictionary inside of a list)

category_creation(*type_ids*)
Method for constructing a list of variable categories, using the jsonstat table retrieved from cantabular.

Return type List[Dict[str, object]]

Returns A list of variable categories.

assign_dimensions(*key*)
Method for using the jsonstat table to construct a list of dimensions, based on the initial query to cantabular.

Parameters **key** (str) – The key in the pyjstat dataframe corresponding with the dimensions/variables to be appended to the dataset.

Return type List[Dict[str, object]]

Returns A list of dataset dimensions.

observations(*dataset_id*)
Method for creating dataset observations.

Parameters **dataset_id** (str) – A valid dataset ID, which must be a nonempty string.

Raises

- **TypeError** – If the *dataset_id* is not a string.
- **ValueError** – If the *dataset_id* is an empty string.

Return type Dict[str, object]

Returns A python dict representing dataset dimensions.

static variable_metadata_request(*uuids_metadata*)
Method for the construction of metadata in a format ready to be transmitted to Nomis.

Parameters **uuids_metadata** (List[*UuidMetadata*]) – A list of namedtuples, each containing a UUID and some metadata associated with that UUID.

Raises **TypeError** – If the uuids_metadata passed is not a list, or any of the elements contained in uuids_metadata is not a valid instance of the UuidMetadata namedtuple.

Return type list

Returns A list of metadata, ready to be sent to the Nomis system.

3.5 Nomis API Connectors (Data and Metadata)

3.5.1 nomis_api_connector.py

class nomis_api_connector.**NomisApiConnector**(*credentials*, *address*, *port=None*, *record_requests=True*)

Bases: *api_connector.ApiConnector*

API Connector class for communicating with Nomis' main API. Provides the functionality of appending and altering datasets and variables on the Nomis database through the Nomis API. This is the primary point of interaction between the utility and the Nomis API, containing methods corresponding with all requests the program needs to make. The class is easily extendable to contain more methods should requirements change.

static validate_ds(*ds*, *id=None*)

Method for validating that a dataset is in the correct format and contains the required information. Validation is done simply by first ensuring the dataset is the correct type (a Python dict), and then by checking the individual keys in the dataset, ensuring they are all in the correct type and format, and, if they aren't optional, that they exist at all.

Parameters

- **ds** (Dict[str, object]) – A dictionary representing a dataset ready to be transmitted to the Nomis server.
- **id** (Optional[str]) – Optionally, can include an id to verify that parameter id matches that of dataset

Raises

- **TypeError** – If the dataset itself or any of its elements are of an incorrect type.
- **KeyError** – If the dataset is missing elements or has unexpected ones.
- **ValueError** – If the dataset contains an uuid and it isn't in a valid UUID format.

Return type bool

Returns If an exception isn't raised, the method will return True.

static validate_id(*id*)

Method for validating parameter IDs.

Parameters **id** (str) – A string that is in a valid ID format.

Raises

- **TypeError** – If the inputted id is not a string.
- **ValueError** – If the inputted id is not in the correct format.

Return type bool

Returns *True* if the id is valid, otherwise an exception is raised.

get_dataset(*id*, *return_bool=False*)

Method for obtaining a dataset from the Nomis database by its uuid. Makes a GET request to the Nomis API at the /Datasets/{id} endpoint, and returns the dataset if the response code is 200; otherwise, an appropriate exception is raised. Alternatively, if the return_bool param is set to True, this method will simply check for the existence of a dataset and return a Boolean response.

Parameters

- **id** (str) – A string that is in a valid ID format.

- **return_bool** (bool) – Admin parameter; returns True or False instead of returning the dataset or raising exception in the case of a 200 or 404 response.

Raises

- **TypeError** – If the validate_id() method detects that the id is not a string.
- **ValueError** – If the validate_id() method detects that the dataset uuid is not in the correct UUID format.
- **requests.HTTPError** – If either connecting to the API is unsuccessful or a negative response is received.

Return type Union[Dict[str, object], bool]

Returns If the request is successful, then return the dataset associated with the inputted ID. Alternatively, if *return_bool* is *True*, then return *True* if the dataset exists, otherwise return *False*.

create_dataset(*id*, *ds*)

Method for uploading a dataset to the Nomis database. Makes a PUT request to the /Datasets/{id} endpoint, with a valid Dataset object encoded into JSON as the body. True is returned by the method if the dataset creation is successful (i.e., a 200 code is received), otherwise an appropriate exception is raised.

Parameters

- **id** (str) – A string that is in a valid id format.
- **ds** (Dict[str, object]) – Valid dict representing a Nomis dataset.

Raises

- **TypeError** – If the validate_ds() or the validate_id() methods.
- **ValueError** – Could be raised by the validate_ds() or the validate_id() methods.
- **requests.HTTPError** – If either connecting to the API is unsuccessful or a negative response is received.

Return type bool

Returns Unless an exception is raised, will return True indicating the request was successful.

get_dataset_dimensions(*id*, *return_bool=False*)

Method for retrieving dataset dimensions for a dataset with the parameter ID. Makes a GET request to the /Datasets/{id}/dimensions endpoint. If successful, then the dimensions are returned by the method, otherwise an appropriate exception is raised.

Parameters

- **id** (str) – A string that is in a valid id format.
- **return_bool** (bool) – Return a bool instead of the dimensions themselves

Raises

- **TypeError** – If the validate_id() method detects that the id is not a string.
- **ValueError** – If the validate_id() method detects that the dataset uuid is not in the correct UUID format.
- **requests.HTTPError** – If either connecting to the API is unsuccessful or a negative response is received.

Return type Union[List[Dict[str, object]], bool]

Returns Unless an exception is raised, will return a list of dimensions for the dataset requested.

assign_dimensions_to_dataset(*id*, *dims*)
Method for assigning dimensions to a dataset which exists in the Nomis database.

Parameters

- **id** (str) – A string that is in a valid id format.
- **dims** (Union[list, dict]) – Object representing the dimensions.

Raises

- **TypeError** – If the validate_id() method detects that the id is not a string.
- **ValueError** – If the validate_id() method detects that the inputted id is not in the correct UUID format.
- **requests.HTTPError** – If either connecting to the API is unsuccessful or a negative response is received.

Return type bool

Returns *True* if dimensions are assigned successfully, otherwise an exception is raised.

append_dataset_observations(*id*, *obs*)
Method for appending observations to a dataset in the database.

Parameters

- **id** (str) – A string that is in a valid id format.
- **obs** (Dict[str, object]) – Object representing observation values.

Raises

- **TypeError** – If the validate_id() method detects that the id is not a string, or due to invalid observations.
- **ValueError** – If the validate_id() method detects that the inputted id is not in the correct UUID format.
- **requests.HTTPError** – If either connecting to the API is unsuccessful or a negative response is received.

Return type bool

Returns Unless an exception is raised, *True* is returned indicating a successful request.

overwrite_dataset_observations(*id*, *obs_arr*)
Method for overwriting the observations of a dataset in the Nomis database.

Parameters

- **id** (str) – A string that is in a valid id format.
- **obs_arr** (Union[list, dict]) – Array of objects representing data values.

Raises

- **TypeError** – If the validate_id() method detects that the id is not a string, or due to invalid observations.
- **ValueError** – If the validate_id() method detects that the inputted id is not in the correct UUID format.
- **requests.HTTPError** – If either connecting to the API is unsuccessful or a negative response is received.

Return type bool

Returns Unless an exception is raised, *True* is returned indicating a successful request.

get_variable(*name=None*, *return_bool=False*)

Method for retrieving an existing variable, or simply checking for its existence and returning a Boolean confirmation.

Parameters

- **name** (Optional[str]) – Unique name of the variable.
- **return_bool** (bool) – Admin parameter; returns False instead of raising an exception if variable not found

Raises

- **TypeError** – If the name param is not a string.
- **ValueError** – If the name param is not in a valid format.
- **requests.HTTPError** – If either connecting to the API is unsuccessful or a negative response is received.

Return type Union[Dict[str, object], List[Dict[str, object]], bool]

Returns Unless an exception is raised, the variable (as a dictionary) is returned.

create_variable(*name*, *var*)

Method for creating a new variable.

Parameters

- **name** (str) – Unique name of the variable.
- **var** (dict) – Object representing the variable. Must be a valid dict.

Raises

- **TypeError** – If the name param is not a string, or the var param is not a valid Variable.
- **ValueError** – If the name param is not in a valid format.
- **requests.HTTPError** – If either connecting to the API is unsuccessful or a negative response is received.

Return type bool

Returns Unless an exception is raised, True is returned if the variable does exist, False otherwise.

get_variable_categories(*name*)

Method for retrieving the categories of a variable by name.

Parameters **name** (str) – Unique name of the variable.

Raises

- **TypeError** – If the name param is not a string.
- **ValueError** – If the name param is not in a valid format.
- **requests.HTTPError** – If either connecting to the API is unsuccessful or a negative response is received.

Return type list

Returns A list of categories upon a successful request; an appropriate error will be raised otherwise.

create_variable_category(*name*, *cat_arr*)
Method for adding variable categories to a variable in the dataset.

Parameters

- **name** (str) – Unique name of the variable.
- **cat_arr** (list) – Array of dimension categories.

Raises

- **TypeError** – If the name param is not a string, or the cat_arr param is not a valid list of categories.
- **ValueError** – If the name param is not in a valid format.
- **requests.HTTPError** – if either connecting to the API is unsuccessful or a negative response is received.

Return type bool

Returns True will be returned upon a successful request; an appropriate error will be raised otherwise.

update_variable_category(*name*, *code*, *cat*)
Method for updating a specific variable category.

Parameters

- **name** (str) – Unique name of the variable.
- **code** (str) – Category code
- **cat** (dict) – Partial object representing a variable category.

Raises

- **TypeError** – If the name param is not a string, the cat param is not a valid category, or the code is invalid.
- **ValueError** – If the name param is not in a valid format.
- **requests.HTTPError** – If either connecting to the API is unsuccessful or a negative response is received.

Return type bool

Returns True will be returned upon a successful request; an appropriate error will be raised otherwise.

create_variable_type(*name*, *type_arr*)
Method for adding variable types to a variable in the dataset.

Parameters

- **name** (str) – Unique name of the variable.
- **type_arr** (list) – Array of dimension types.

Raises

- **TypeError** – If the name param is not a string, or the type_arr param is not a valid list of types.
- **ValueError** – If the name param is not in a valid format.
- **requests.HTTPError** – if either connecting to the API is unsuccessful or a negative response is received.

Return type bool

Returns True will be returned upon a successful request; an appropriate error will be raised otherwise.

update_variable_type(*variable_id*, *type_id*, *var_type*)
Method for updating a variable type for a variable in the dataset.

Parameters

- **variable_id** (str) – Unique ID of the variable.
- **type_id** (str) – The ID of the specific type.
- **var_type** (dict) – Dimension type.

Raises

- **TypeError** – If the variable_id param is not a string, or the var_type param is not a valid dict, or the type_id param is not a string.
- **ValueError** – If the variable_id or type_id param is not in a valid format.
- **requests.HTTPError** – if either connecting to the API is unsuccessful or a negative response is received.

Return type bool

Returns True will be returned upon a successful request; an appropriate error will be raised otherwise.

3.5.2 nomis_metadata_api_connector.py

class nomis_metadata_api_connector.**NomisMetadataApiConnector**(*credentials*, *address*, *port=None*, *record_requests=None*)

Bases: *api_connector.ApiConnector*

Class for communicating directly with the Nomis metadata API; includes methods for retrieving metadata (by object or metadata ID), adding new/overwriting metadata, and updating existing metadata. This is the central hub of communication between the Nomis metadata API and the utility. It is easily extendable to contain more methods should the requirements change necessitating additional requests.

static validate_uuid(*id*)
Method for validating the id of a dataset.

Raises

- **TypeError** – If the dataset's id is not a string.
- **ValueError** – If the dataset is not in a valid uuid format.

Return type bool

Returns *True* if the id is valid, otherwise an exception is raised.

static validate_metadata(*metadata*, *id=None*)
Method for validating metadata, in terms of format and contents.

Raises

- **TypeError** – If the metadata is not in the correct type (i.e., a Python dict), or if validate_uuid() detects a type error.
- **ValueError** – If validate_uuid() detects a value error.

Return type `bool`

Returns *True* if the metadata is valid, otherwise an exception is raised.

get_all_metadata()
Method to retrieve all of the metadata on the server.

Raises

- **requests.ConnectionError** – If an error occurs whilst attempting to communicate with the API.
- **requests.HTTPError** – If a negative response is received from the API.

Return type `List[Dict[str, Union[str, List[str]]]]`

Returns A list of metadata, if the request is a success; otherwise, an exception will have been raised.

get_metadata_for_object(*id*, *return_bool=False*)
This takes a uuid representing the id of an object in the database as a parameter, and it makes a GET request to retrieve the metadata associated with this object.

Parameters

- **id** (`str`) – Valid string in uuid format representing the id of an object in the Nomis database.
- **return_bool** (`bool`) – Boolean toggle where, if set to True, will force the method to return a Boolean instead of the metadata.

Raises

- **requests.ConnectionError** – If an error occurs whilst attempting to communicate with the API.
- **requests.HTTPError** – If a negative response is received from the API.

Return type `Union[List[Dict[str, Union[str, List[str]]]], bool]`

Returns The metadata for the object with the inputted id, if it exists; otherwise, an exception will be raised. Alternatively, if *return_bool* is set to *True*, return True if the metadata exists and False otherwise.

get_metadata_by_id(*id*, *return_bool=False*)
This takes a uuid representing the id of some metadata in the database as a parameter, and makes a GET request to retrieve this metadata. NOTE that this is strictly distinct from the previous method: the uuid of metadata is not necessarily the same as the uuid of the object it represents (and some metadata represents no object).

Parameters

- **id** (`str`) – Valid string representing the ID of some metadata in the Nomis database.
- **return_bool** (`bool`) – Forces method to return True or False instead of returning the metadata or raising an exception in the case of a 200 or 404 response.

Return type `Union[Dict[str, Union[str, List[str]]], bool]`

Returns *False* if the id doesn't exist, is invalid, or has no associated metadata; the metadata, otherwise

add_new_metadata(*metadata*, *return_uuids=False*)
This takes an object representing an instance of Metadata and makes a POST request that adds this metadata to the server. This metadata is not required to have anything for its id: if it has no id then the API will

generate a uuid for it automatically. If it does have an id, it must be in a valid uuid format, if not then the server will respond with an error (I have added a method to verify the uuid before the request is made). As a warning, the server will OVERWRITE any metadata on the server that has the same id as what is being posted. Therefore, calling the get_metadata_by_id() method prior to this method is worthwhile to check whether we will or won't overwrite anything by calling this method. This method returns the uuid of the metadata that was appended to the server.

Parameters

- **metadata** (`List[Dict[str, Union[str, List[str]]]]`) – Valid list of dictionary of strings representing metadata.
- **return_uuids** (`bool`) – Toggle for returning uuids instead of a boolean confirmation - for testing purposes.

Return type `Union[List[str], bool]`

Returns Bool indicating the success of the request, or the ids of the appended datasets if toggled for.

update_metadata_association(*id*, *metadata*)

As above, this method takes an instance of Metadata as a parameter, but it also requires a valid uuid as an additional parameter. This method entails making a PUT request that will update any existing metadata, or create some new metadata if the id does not exist on the server. The metadata object here can be "incomplete", in which case it will only update the included fields upon making the request; however, the metadata must have a valid id, and this id must match the one passed to the method in the parameters.

Parameters

- **id** (`str`) – Valid string representing the ID of some metadata in the Nomis database.
- **metadata** (`Dict[str, Union[str, List[str]]]`) – Valid dictionary of strings representing metadata attributes (must include belongsTo).

Return type `bool`

Returns Bool indicating the success of the request.

3.6 Additional Modules and Parent Classes

3.6.1 api_connector.py

class api_connector.**ApiConnector**(*credentials*, *address*, *port*, *record_requests=True*)

Bases: `object`

Parent class for the three Api Connectors (namely, CantabularApiConnector, NomisApiConnector, and NomisMetadataApiConnector).

Parameters

- **address** (`str`) – A string of a valid address, either a url or IP address, for connecting to the API.
- **credentials** (`Tuple[str, str]`) – Contains the username and password for authentication with the API.
- **port** (`Union[str, int, None]`) – A string or an integer representing the port the API will be served on.

Variables

- **client** (*str*) – Concatenation of the address and the port, if a port is included; otherwise, just the address.
- **session** (*Session*) – A requests Session instance with authorisation for the associated API.
- **record_requests** (*bool*) – Boolean toggle, when set to *True* the save_request() method will permitted, whereas when set to *False*, it will be prohibited.

save_request(*method*, *res*)
Method for saving requests made and their responses (success or failure) to distinct files. Each created file will reside within a directory specific to this particular API connector instance.

Parameters

- **method** (str) – The connector method within which the request was made.
- **res** (Optional[Response]) – The response received by the system.

Return type None

3.6.2 file_reader.py

class file_reader.**FileReader**(*file*)
Bases: object

Class for handling files; reading, writing, and checking for existence.

Parameters **file** (str) – A string representing the location of a file for use in the program.

Ivar Initial value: file.

exists()
Ensure the file exists by checking the path is an existing one.

Raises **FileNotFoundError** – If the file attribute contains a string that represents a path which doesn't exist.

Return type bool

Returns *True* if no exception is raised, indicating that the file is an existing one.

load_json()
Check the encoding of the file and load it in as a json string.

Return type dict

write_json(*to_write*)
Write out a json file.

Return type None

write_text_file(*to_write*)
Write out a text file.

Return type None

3.6.3 type_hints.py

class type_hints.**UuidMetadata**(*uuid*, *metadata*)
Bases: tuple

property metadata
Alias for field number 1

property uuid
Alias for field number 0

class type_hints.**CredentialsConninfo**(*credentials*, *connection_info*)
Bases: tuple

property connection_info
Alias for field number 1

property credentials
Alias for field number 0

APPENDIX OF ARGUMENTS

-c

Config file [string] : Supply a path for a config file, which will overwrite the default path within the following run and force the program to try to use the data within this config file.

-d

Query dataset [string] : The census dataset to be retrieved from the Cantabular system.

-db

Debug [toggle] : When this flag is enabled, the "debug" mode is toggled, entailing a very detailed output to stdout throughout the run.

-f

Filename [string] : Read data from a file instead of querying cantabular.

-h

Help [toggle] : This prevents the utility from running. A help message is printed to stdout detailing the arguments you see here and their usage.

-i

Dataset ID [string] : The ID of the Nomis dataset to be updated or created.

-l

Log file [string] : Supply a path for a log file, which will overwrite the default log path within the following run and force the program to log to this file instead.

-q

Query variables [string] : Delimited list input e.g. "COUNTRY, SEX".

-r

Metadata format [string] : The format of metadata that the program should expect: i.e., 'C' $\sim$ Cantabular, or 'O' $\sim$ ONS.

-t

Dataset title [string] : The title of the Nomis dataset to be updated or created (if updating the dataset, this title will overwrite the current title).

transformation

Transformation mode [string] : This argument has no flag, instead it is a positional that immediately follows main.py. It can be either *data* or *metadata*, and it dictates the mode of the program for the followin run.

-v

Verbose [toggle] : When this flag is enabled, the "verbose" mode is toggled, meaning that there will be a verbose output to stdout throughout the following run.

-y

Suppress prompts [toggle] : When this flag is enabled, the "suppress all prompts" mode is toggled, meaning that "yes" will be automatically responded to all (y/n) prompts.

APPENDIX OF EXCEPTIONS

FileNotFoundError

A FileNotFoundError will be raised by the `FileReader` at any point it receives a file as its parameter that it is unable to locate.

IOError

An IOError is raised if a certain file inputted in the arguments (i.e., a config file or a log file) is of an invalid or unexpected type (for instance, if the config file is not a JSON).

KeyError

A KeyError is raised by the Nomis API Connector if, when validating the dataset, it detects insufficient keys (i.e., expected ones are non-existing), or when the Geography variables haven't been defined in the configuration, and these are attempted to be retrieved.

LookupError

The program raises a LookupError if it attempts to access expected keys in the table instance variable of `DatasetTransformations` and fails to do so.

NameError

A NameError is raised if the configuration details (i.e., the credentials or client) are attempted to be retrieved for an API that has not been established in the configuration file.

requests.ConnectionError

The program will raise a ConnectionError (from the requests library) at any point it is unable to successfully establish a connection with one of the APIs.

requests.HTTPError

The program will raise an HTTPError (from the requests library) at any point it receives a response from one of the APIs where the status code is greater and or equal to 400, which indicates a bad request.

TypeError

The program will raise a TypeError at any point where it is validating some parameter or argument and detects an invalid type.

ValueError

The program will raise a ValueError at any point where an argument or parameter has an invalid value, for instance it's numeric value is out of range, it's an empty string, it doesn't correctly resolve to a UUID or IP address, and so on.

INDEX OF MODULES