ISA tools API Documentation

Release 1.0

ISA Team

Jul 07, 2017

Contents

The ISA tools API is published on PyPI as the `isatools` Python package. The package aims to provide you, the developer, with a set of tools to help you easily and quickly build your own ISA objects, validate, and convert between serializations of ISA-formatted datasets and other formats/schemas (e.g. SRA schemas). The goal of this package is to provide a flexible way to build and use ISA content, as well as provide utility functions for file conversions and validation.

Note: `isatools` is currently only supported in Python 3.4 and 3.5. Python 2.7 support is present in the `py2` source code branch in Github.

CHAPTER 1

License

This code is licensed under the CPAL License.

Installation

Requires: Python 3.4 or 3.5; additionally Java 1.6+ for validator and SRA conversion

Installation from the Python Package Index

The ISA API is published on the Python Package Index (PyPI) as the *isatools* package ((see https://pypi.python.org/pypi/isatools/), and you can use `pip` to install it.

```
$ pip install isatools
```

Now you're ready to get started!

Installation from sources

The ISA-API source code is hosted on GitHub at: https://github.com/ISA-tools/isa-api

We recommend using a virtual environment for your Python projects. `virtualenv` is a tool for creating isolated Python runtime environments. It does not install separate copies of Python, but rather it does provide a clever way to keep different configurations of environment cleanly separated.

If you are on Mac OS X or Linux, one of the following two commands will work for you:

```
$ sudo easy_install virtualenv
```

or even better:

```
$ sudo pip install virtualenv
```

Then, you can create a virtual environament: `$ virtualenv venv`

and activate it: `$ source venv/bin/activate`

Finally, you should install the requirements with: `$ pip install -r requirements.txt`

or

`$ pip install --upgrade -r requirements.txt` if you want to upgrade the requirements already installed.

Now you're ready to get started!

For full instructions on installing and using `virtualenv` see their documentation.

Running tests

The tests in the ISA-API rely on datasets available in the test branch of the ISAdatasets repository.

Thus, the first step for running the tests is to clone that branch under the /tests/data folder:

```
git clone -b tests --single-branch http://github.com/ISA-tools/ISAdatasets
tests/data
```

After that, you can run the test with the usual command:

```
python setup.py test
```

Logging

By default the ISA-API will output error messages to the standard output. To control the logging level, you can set the logging level via the *isatools* package as follows:

‘‘‘ import isatools import logging isatools.log_level = logging.INFO # sets log level to INFO

from isatools import isatab “”” Now do some stuff with the isatab package - logging should output messages at INFO, WARNING, ERROR and FATAL levels “”” ISA = isatab.load(...)

‘‘‘ Note that you cannot reset the log level interactively after setting it the first time, so you would need to reload your environment (iPython etc.) to change the log level again.

ISA data model

For the ISA tools API, we have represented the ISA model version 1.0 (see the ISA-Tab specification) with a set of JSON schemas, which provide the information the ISA model maintains for each of the objects.

The objective of designing and developing JSON schemas is to support a new serialization of the ISA-Tab model in JSON format, in addition to existing serializations in tabular format and RDF format.

The core set of schemas for ISA model version 1.0 can be found in the folder isatools/schemas/isa_model_version_1_0_schemas/core.

The main object is the Investigation, which groups a set of Studies and maintains associated information such as Publications, People involved and the ontologies used for annotating the dataset.

Creating ISA content

The ISA API provides a set of Python classes that you can use to create ISA content with.

The three main objects that you need to create ISA content are:

- `Investigation`
- `Study`
- `Assay`

...of course!

Important: As a pre-requisite to using ISA model classes, please make sure you have read and understood the *ISA Abstract Model* that the ISA formats are based on.

Getting started

In `isatools.model.v1`, the class `Investigation` is used as the top level container for all other ISA content. The `Investigation` Python class corresponds to the Investigation as defined in the ISA Model Specification. For example, to create an empty ISA structure consisting of an investigation with one study, you might use the following code:

```
>>> from isatools.model.v1 import *
>>> investigation = Investigation()
>>> investigation.studies.append(Study())  # adds a new default Study object to␣
↪investigation
```

This code simply creates one `Investigation` object, and adds a single `Study` object to it. The constructor of each of these objects creates empty structures for each of these. We can then inspect the structure by accessing its instance variables as follows:

```
>>> investigation.studies
[<isatools.model.v1.Study object>]

>>> investigation.studies[0].assays
[]

>>> investigation.title
''
```

Since we have not set any data in our ISA objects, these are by default mostly empty at the moment. We can set some instance variables with data as follows:

```
>>> investigation.title = "My ISA Investigation"
>>> investigation.title
'My ISA Investigation'

>>> investigation.studies[0].title = "My ISA Study"
>>> investigation.studies[0].title
'My ISA Study'

>>> investigation.studies[0].assays.append(Assay())  # adds a new default Assay␣
↪object to study
```

```
>>> i.studies[0].assays
[<isatools.model.v1.Assay object>]
```

If you then write these out to ISA-Tab, we can inspect the output written into an `i_investigation.txt` file. We do this using the `isatab` module to `dump()` the `Investigation` object we created, as follows:

```
>>> from isatools import isatab
>>> isatab.dump(investigation, 'tmp/')  # dump out ISA-Tab  to tmp/
<isatools.model.v1.Investigation object>
```

If all went as expected, you should find an `i_investigation.txt` file with the standard Investigation sections, one Study section structured as defined by the ISA-Tab Specification.

Hint: Remember that when you `dump()` ISA content, you do it on the `Investigation` object. This means any `Study` and `Assay` objects and content must be attached to the `Investigation` for it to be serialized out.

Different classes in `isatools.model.v1` have class constructors and instance variables that roughly map to the ISA Abstract Model. For full details of how to instantiate model classes, access and manipulate ISA data as objects, please inspect the module's docstrings.

Obviously this isn't enough to create a fully populated ISA investigation, but we would recommend that you have a look in the `isatools.model.v1` package to inspect all the docstring documentation that is included with each of the ISA model classes.

Tutorial: describing a simple experiment

In this section, we provide a basic example of creating a complete experiment descriptor using the ISA API's model classes. The descriptor is not complete and realistic, but it demonstrates the range of component classes that you can use to create ISA content, including things like sample characteristics, ontology annotations and units.

Important: As a pre-requisite to using ISA model please make sure you have read and understood the *ISA Abstract Model* that the ISA formats are based on.

Firstly, we need to import the ISA API's model classes from the `isatools` PyPI package.

```
from isatools.model.v1 import *
```

Next, we build our descriptor encapsulated in a single Python function to simplify the example code. In a real application or script, you might decompose the functionality and hook it up to interactive components to solicit feedback from a user on-the-fly.

```
def create_descriptor():
    """Returns a simple but complete ISA-Tab 1.0 descriptor for illustration."""

    # Create an empty Investigation object and set some values to the instance␣
↪variables.

    investigation = Investigation()
    investigation.identifier = "i1"
    investigation.title = "My Simple ISA Investigation"
    investigation.description = "We could alternatively use the class constructor's␣
↪parameters to set some default " \
```

```
                                "values at the time of creation, however we want to␣
↪demonstrate how to use the " \
                                "object's instance variables to set values."
    investigation.submission_date = "2016-11-03"
    investigation.public_release_date = "2016-11-03"

    # Create an empty Study object and set some values. The Study must have a␣
↪filename, otherwise when we serialize it
    # to ISA-Tab we would not know where to write it. We must also attach the study␣
↪to the investigation by adding it
    # to the 'investigation' object's list of studies.

    study = Study(filename="s_study.txt")
    study.identifier = "s1"
    study.title = "My ISA Study"
    study.description = "Like with the Investigation, we could use the class␣
↪constructor to set some default values, " \
                        "but have chosen to demonstrate in this example the use of␣
↪instance variables to set initial " \
                        "values."
    study.submission_date = "2016-11-03"
    study.public_release_date = "2016-11-03"
    investigation.studies.append(study)

    # Some instance variables are typed with different objects and lists of objects.␣
↪For example, a Study can have a
    # list of design descriptors. A design descriptor is an Ontology Annotation␣
↪describing the kind of study at hand.
    # Ontology Annotations should typically reference an Ontology Source. We␣
↪demonstrate a mix of using the class
    # constructors and setting values with instance variables. Note that the␣
↪OntologyAnnotation object
    # 'intervention_design' links its 'term_source' directly to the 'obi' object␣
↪instance. To ensure the OntologySource
    # is encapsulated in the descriptor, it is added to a list of 'ontology_source_
↪references' in the Investigation
    # object. The 'intervention_design' object is then added to the list of 'design_
↪descriptors' held by the Study
    # object.

    obi = OntologySource(name='OBI', description="Ontology for Biomedical␣
↪Investigations")
    investigation.ontology_source_references.append(obi)
    intervention_design = OntologyAnnotation(term_source=obi)
    intervention_design.term = "intervention design"
    intervention_design.term_accession = "http://purl.obolibrary.org/obo/OBI_0000115"
    study.design_descriptors.append(intervention_design)

    # Other instance variables common to both Investigation and Study objects include
↪'contacts' and 'publications',
    # each with lists of corresponding Person and Publication objects.

    contact = Person(first_name="Alice", last_name="Robertson", affiliation=
↪"University of Life", roles=[OntologyAnnotation(term='submitter')])
    study.contacts.append(contact)
    publication = Publication(title="Experiments with Elephants", author_list="A.␣
↪Robertson, B. Robertson")
    publication.pubmed_id = "12345678"
```

```
    publication.status = OntologyAnnotation(term="published")
    study.publications.append(publication)

    # To create the study graph that corresponds to the contents of the study table␣
↪file (the s_*.txt file), we need
    # to create a process sequence. To do this we use the Process class and attach it␣
↪to the Study object's
    # 'process_sequence' list instance variable. Each process must be linked with a␣
↪Protocol object that is attached to
    # a Study object's 'protocols' list instance variable. The sample collection␣
↪Process object usually has as input
    # a Source material and as output a Sample material.

    # Here we create one Source material object and attach it to our study.

    source = Source(name='source_material')
    study.materials['sources'].append(source)

    # Then we create three Sample objects, with organism as Homo Sapiens, and attach␣
↪them to the study. We use the utility function
    # batch_create_material() to clone a prototype material object. The function␣
↪automatiaclly appends
    # an index to the material name. In this case, three samples will be created,␣
↪with the names
    # 'sample_material-0', 'sample_material-1' and 'sample_material-2'.

    prototype_sample = Sample(name='sample_material', derives_from=source)
    ncbitaxon = OntologySource(name='NCBITaxon', description="NCBI Taxonomy")
    characteristic_organism = Characteristic(category=OntologyAnnotation(term=
↪"Organism"),
                                     value=OntologyAnnotation(term="Homo Sapiens",␣
↪term_source=ncbitaxon,
                                                              term_accession="http://
↪purl.bioontology.org/ontology/NCBITAXON/9606"))
    prototype_sample.characteristics.append(characteristic_organism)

    study.materials['samples'] = batch_create_materials(prototype_sample, n=3)  #␣
↪creates a batch of 3 samples

    # Now we create a single Protocol object that represents our sample collection␣
↪protocol, and attach it to the
    # study object. Protocols must be declared before we describe Processes, as a␣
↪processing event of some sort
    # must execute some defined protocol. In the case of the class model, Protocols␣
↪should therefore be declared
    # before Processes in order for the Process to be linked to one.

    sample_collection_protocol = Protocol(name="sample collection",
                                          protocol_type=OntologyAnnotation(term=
↪"sample collection"))
    study.protocols.append(sample_collection_protocol)
    sample_collection_process = Process(executes_protocol=sample_collection_protocol)

    # Next, we link our materials to the Process. In this particular case, we are␣
↪describing a sample collection
    # process that takes one source material, and produces three different samples.
    #
    # (source_material)->(sample collection)->[(sample_material-0), (sample_material-
↪1), (sample_material-2)]
```

```
    for src in study.materials['sources']:
        sample_collection_process.inputs.append(src)
    for sam in study.materials['samples']:
        sample_collection_process.outputs.append(sam)

    # Finally, attach the finished Process object to the study process_sequence. This␣
↪can be done many times to
    # describe multiple sample collection events.

    study.process_sequence.append(sample_collection_process)

    # Next, we build n Assay object and attach two protocols, extraction and␣
↪sequencing.

    assay = Assay(filename="a_assay.txt")
    extraction_protocol = Protocol(name='extraction', protocol_
↪type=OntologyAnnotation(term="material extraction"))
    study.protocols.append(extraction_protocol)
    sequencing_protocol = Protocol(name='sequencing', protocol_
↪type=OntologyAnnotation(term="material sequencing"))
    study.protocols.append(sequencing_protocol)

    # To build out assay graphs, we enumereate the samples from the study-level, and␣
↪for each sample we create an
    # extraction process and a sequencing process. The extraction process takes as␣
↪input a sample material, and produces
    # an extract material. The sequencing process takes the extract material and␣
↪produces a data file. This will
    # produce three graphs, from sample material through to data, as follows:
    #
    # (sample_material-0)->(extraction)->(extract-0)->(sequencing)->(sequenced-data-0)
    # (sample_material-1)->(extraction)->(extract-1)->(sequencing)->(sequenced-data-1)
    # (sample_material-2)->(extraction)->(extract-2)->(sequencing)->(sequenced-data-2)
    #
    # Note that the extraction processes and sequencing processes are distinctly␣
↪separate instances, where the three
    # graphs are NOT interconnected.

    for i, sample in enumerate(study.materials['samples']):

        # create an extraction process that executes the extraction protocol

        extraction_process = Process(executes_protocol=extraction_protocol)

        # extraction process takes as input a sample, and produces an extract␣
↪material as output

        extraction_process.inputs.append(sample)
        material = Material(name="extract-{}".format(i))
        material.type = "Extract Name"
        extraction_process.outputs.append(material)

        # create a sequencing process that executes the sequencing protocol

        sequencing_process = Process(executes_protocol=sequencing_protocol)
        sequencing_process.name = "assay-name-{}".format(i)
        sequencing_process.inputs.append(extraction_process.outputs[0])
```

```
        # Sequencing process usually has an output data file

        datafile = DataFile(filename="sequenced-data-{}".format(i), label="Raw Data␣
↪File")
        sequencing_process.outputs.append(datafile)

        # Ensure Processes are linked forward and backward. plink(from_process, to_
↪process) is a function to set
        # these links for you. It is found in the isatools.model.v1 package

        plink(extraction_process, sequencing_process)

        # make sure the extract, data file, and the processes are attached to the␣
↪assay

        assay.data_files.append(datafile)
        assay.materials['other_material'].append(material)
        assay.process_sequence.append(extraction_process)
        assay.process_sequence.append(sequencing_process)
        assay.measurement_type = OntologyAnnotation(term="gene sequencing")
        assay.technology_type = OntologyAnnotation(term="nucleotide sequencing")

    # attach the assay to the study

    study.assays.append(assay)
```

To write out the ISA-Tab, you can use the `isatab.dumps()` function:

```
from isatools.isatab import dumps
return dumps(investigation)  # dumps() writes out the ISA as a string representation␣
↪of the ISA-Tab
```

The function listed above is designed to return all three files as a single string output for ease of inspection. Alternatively you could do something like `dump(isa_obj=investigation, output_path='./')` to write the files to the file system.

Alternatively to write out the ISA JSON, you can use the `ISAJSONEncoder` class with the Python `json` package:

```
import json
from isatools.isajson import ISAJSONEncoder
# Note that the extra parameters sort_keys, indent and separators are to make the␣
↪output more human-readable.
return json.dumps(investigation, cls=ISAJSONEncoder, sort_keys=True, indent=4,␣
↪separators=(',', ': '))
```

The final lines of code is a `main` routine to invoke the `create_descriptor()` function.

```
if __name__ == '__main__':
    print(create_descriptor())
```

If you save the file into something like `createSimpleISA.py`, to execute this script on the command line and view the output, you would run it with:

```
python createSimpleISA.py
```

This example can be found in the `isatools.examples` package in createSimpleISAtab.py and createSimpleISAJSON.py

createSimpleISAtab.py

An example of using the ISA model classes to create an ISA-Tab set of files.

```
#!/usr/bin/env python

from isatools.model.v1 import *


def create_descriptor():
    """Returns a simple but complete ISA-Tab 1.0 descriptor for illustration."""

    # Create an empty Investigation object and set some values to the instance␣
↪variables.

    investigation = Investigation()
    investigation.identifier = "i1"
    investigation.title = "My Simple ISA Investigation"
    investigation.description = "We could alternatively use the class constructor's␣
↪parameters to set some default " \
                                "values at the time of creation, however we want to␣
↪demonstrate how to use the " \
                                "object's instance variables to set values."
    investigation.submission_date = "2016-11-03"
    investigation.public_release_date = "2016-11-03"

    # Create an empty Study object and set some values. The Study must have a␣
↪filename, otherwise when we serialize it
    # to ISA-Tab we would not know where to write it. We must also attach the study␣
↪to the investigation by adding it
    # to the 'investigation' object's list of studies.

    study = Study(filename="s_study.txt")
    study.identifier = "s1"
    study.title = "My ISA Study"
    study.description = "Like with the Investigation, we could use the class␣
↪constructor to set some default values, " \
                        "but have chosen to demonstrate in this example the use of␣
↪instance variables to set initial " \
                        "values."
    study.submission_date = "2016-11-03"
    study.public_release_date = "2016-11-03"
    investigation.studies.append(study)

    # Some instance variables are typed with different objects and lists of objects.␣
↪For example, a Study can have a
    # list of design descriptors. A design descriptor is an Ontology Annotation␣
↪describing the kind of study at hand.
    # Ontology Annotations should typically reference an Ontology Source. We␣
↪demonstrate a mix of using the class
    # constructors and setting values with instance variables. Note that the␣
↪OntologyAnnotation object
    # 'intervention_design' links its 'term_source' directly to the 'obi' object␣
↪instance. To ensure the OntologySource
    # is encapsulated in the descriptor, it is added to a list of 'ontology_source_
↪references' in the Investigation
    # object. The 'intervention_design' object is then added to the list of 'design_
↪descriptors' held by the Study
```

```
    # object.

    obi = OntologySource(name='OBI', description="Ontology for Biomedical␣
↪Investigations")
    investigation.ontology_source_references.append(obi)
    intervention_design = OntologyAnnotation(term_source=obi)
    intervention_design.term = "intervention design"
    intervention_design.term_accession = "http://purl.obolibrary.org/obo/OBI_0000115"
    study.design_descriptors.append(intervention_design)

    # Other instance variables common to both Investigation and Study objects include
↪'contacts' and 'publications',
    # each with lists of corresponding Person and Publication objects.

    contact = Person(first_name="Alice", last_name="Robertson", affiliation=
↪"University of Life", roles=[OntologyAnnotation(term='submitter')])
    study.contacts.append(contact)
    publication = Publication(title="Experiments with Elephants", author_list="A.␣
↪Robertson, B. Robertson")
    publication.pubmed_id = "12345678"
    publication.status = OntologyAnnotation(term="published")
    study.publications.append(publication)

    # To create the study graph that corresponds to the contents of the study table␣
↪file (the s_*.txt file), we need
    # to create a process sequence. To do this we use the Process class and attach it␣
↪to the Study object's
    # 'process_sequence' list instance variable. Each process must be linked with a␣
↪Protocol object that is attached to
    # a Study object's 'protocols' list instance variable. The sample collection␣
↪Process object usually has as input
    # a Source material and as output a Sample material.

    # Here we create one Source material object and attach it to our study.

    source = Source(name='source_material')
    study.materials['sources'].append(source)

    # Then we create three Sample objects, with organism as Homo Sapiens, and attach␣
↪them to the study. We use the utility function
    # batch_create_material() to clone a prototype material object. The function␣
↪automatically appends
    # an index to the material name. In this case, three samples will be created,␣
↪with the names
    # 'sample_material-0', 'sample_material-1' and 'sample_material-2'.

    prototype_sample = Sample(name='sample_material', derives_from=[source])
    ncbitaxon = OntologySource(name='NCBITaxon', description="NCBI Taxonomy")
    characteristic_organism = Characteristic(category=OntologyAnnotation(term=
↪"Organism"),
                                             value=OntologyAnnotation(term="Homo Sapiens",␣
↪term_source=ncbitaxon,
                                                                      term_accession="http://
↪purl.bioontology.org/ontology/NCBITAXON/9606"))
    prototype_sample.characteristics.append(characteristic_organism)

    study.materials['samples'] = batch_create_materials(prototype_sample, n=3)  #␣
↪creates a batch of 3 samples
```

```
    # Now we create a single Protocol object that represents our sample collection␣
↪protocol, and attach it to the
    # study object. Protocols must be declared before we describe Processes, as a␣
↪processing event of some sort
    # must execute some defined protocol. In the case of the class model, Protocols␣
↪should therefore be declared
    # before Processes in order for the Process to be linked to one.

    sample_collection_protocol = Protocol(name="sample collection",
                                          protocol_type=OntologyAnnotation(term=
↪"sample collection"))
    study.protocols.append(sample_collection_protocol)
    sample_collection_process = Process(executes_protocol=sample_collection_protocol)

    # Next, we link our materials to the Process. In this particular case, we are␣
↪describing a sample collection
    # process that takes one source material, and produces three different samples.
    #
    # (source_material)->(sample collection)->[(sample_material-0), (sample_material-
↪1), (sample_material-2)]

    for src in study.materials['sources']:
        sample_collection_process.inputs.append(src)
    for sam in study.materials['samples']:
        sample_collection_process.outputs.append(sam)

    # Finally, attach the finished Process object to the study process_sequence. This␣
↪can be done many times to
    # describe multiple sample collection events.

    study.process_sequence.append(sample_collection_process)

    # Next, we build n Assay object and attach two protocols, extraction and␣
↪sequencing.

    assay = Assay(filename="a_assay.txt")
    extraction_protocol = Protocol(name='extraction', protocol_
↪type=OntologyAnnotation(term="material extraction"))
    study.protocols.append(extraction_protocol)
    sequencing_protocol = Protocol(name='sequencing', protocol_
↪type=OntologyAnnotation(term="material sequencing"))
    study.protocols.append(sequencing_protocol)

    # To build out assay graphs, we enumereate the samples from the study-level, and␣
↪for each sample we create an
    # extraction process and a sequencing process. The extraction process takes as␣
↪input a sample material, and produces
    # an extract material. The sequencing process takes the extract material and␣
↪produces a data file. This will
    # produce three graphs, from sample material through to data, as follows:
    #
    # (sample_material-0)->(extraction)->(extract-0)->(sequencing)->(sequenced-data-0)
    # (sample_material-1)->(extraction)->(extract-1)->(sequencing)->(sequenced-data-1)
    # (sample_material-2)->(extraction)->(extract-2)->(sequencing)->(sequenced-data-2)
    #
    # Note that the extraction processes and sequencing processes are distinctly␣
↪separate instances, where the three
```

```
    # graphs are NOT interconnected.

    for i, sample in enumerate(study.materials['samples']):

        # create an extraction process that executes the extraction protocol

        extraction_process = Process(executes_protocol=extraction_protocol)

        # extraction process takes as input a sample, and produces an extract␣
↪material as output

        extraction_process.inputs.append(sample)
        material = Material(name="extract-{}".format(i))
        material.type = "Extract Name"
        extraction_process.outputs.append(material)

        # create a sequencing process that executes the sequencing protocol

        sequencing_process = Process(executes_protocol=sequencing_protocol)
        sequencing_process.name = "assay-name-{}".format(i)
        sequencing_process.inputs.append(extraction_process.outputs[0])

        # Sequencing process usually has an output data file

        datafile = DataFile(filename="sequenced-data-{}".format(i), label="Raw Data␣
↪File", generated_from=[sample])
        sequencing_process.outputs.append(datafile)

        # Ensure Processes are linked forward and backward. plink(from_process, to_
↪process) is a function to set
        # these links for you. It is found in the isatools.model.v1 package

        plink(extraction_process, sequencing_process)

        # make sure the extract, data file, and the processes are attached to the␣
↪assay

        assay.data_files.append(datafile)
        assay.materials['other_material'].append(material)
        assay.process_sequence.append(extraction_process)
        assay.process_sequence.append(sequencing_process)
        assay.measurement_type = OntologyAnnotation(term="gene sequencing")
        assay.technology_type = OntologyAnnotation(term="nucleotide sequencing")

    # attach the assay to the study

    study.assays.append(assay)

    from isatools.isatab import dumps
    return dumps(investigation)  # dumps() writes out the ISA as a string␣
↪representation of the ISA-Tab

if __name__ == '__main__':
    print(create_descriptor())  # print the result to stdout
```

createSimpleISAjson.py

An example of using the ISA model classes to create an ISA-JSON file.

```
#!/usr/bin/env python

from isatools.model.v1 import *


def create_descriptor():
    """Returns a simple but complete ISA-JSON 1.0 descriptor for illustration."""

    # Create an empty Investigation object and set some values to the instance␣
↪variables.

    investigation = Investigation()
    investigation.identifier = "i1"
    investigation.title = "My Simple ISA Investigation"
    investigation.description = "We could alternatively use the class constructor's␣
↪parameters to set some default " \
                                "values at the time of creation, however we want to␣
↪demonstrate how to use the " \
                                "object's instance variables to set values."
    investigation.submission_date = "2016-11-03"
    investigation.public_release_date = "2016-11-03"

    # Create an empty Study object and set some values. The Study must have a␣
↪filename, otherwise when we serialize it
    # to ISA-Tab we would not know where to write it. We must also attach the study␣
↪to the investigation by adding it
    # to the 'investigation' object's list of studies.

    study = Study(filename="s_study.txt")
    study.identifier = "s1"
    study.title = "My ISA Study"
    study.description = "Like with the Investigation, we could use the class␣
↪constructor to set some default values, " \
                        "but have chosen to demonstrate in this example the use of␣
↪instance variables to set initial " \
                        "values."
    study.submission_date = "2016-11-03"
    study.public_release_date = "2016-11-03"
    investigation.studies.append(study)

    # Some instance variables are typed with different objects and lists of objects.␣
↪For example, a Study can have a
    # list of design descriptors. A design descriptor is an Ontology Annotation␣
↪describing the kind of study at hand.
    # Ontology Annotations should typically reference an Ontology Source. We␣
↪demonstrate a mix of using the class
    # constructors and setting values with instance variables. Note that the␣
↪OntologyAnnotation object
    # 'intervention_design' links its 'term_source' directly to the 'obi' object␣
↪instance. To ensure the OntologySource
    # is encapsulated in the descriptor, it is added to a list of 'ontology_source_
↪references' in the Investigation
    # object. The 'intervention_design' object is then added to the list of 'design_
↪descriptors' held by the Study
```

```
    # object.

    obi = OntologySource(name='OBI', description="Ontology for Biomedical␣
↪Investigations")
    investigation.ontology_source_references.append(obi)
    intervention_design = OntologyAnnotation(term_source=obi)
    intervention_design.term = "intervention design"
    intervention_design.term_accession = "http://purl.obolibrary.org/obo/OBI_0000115"
    study.design_descriptors.append(intervention_design)

    # Other instance variables common to both Investigation and Study objects include
↪'contacts' and 'publications',
    # each with lists of corresponding Person and Publication objects.

    contact = Person(first_name="Alice", last_name="Robertson", affiliation=
↪"University of Life", roles=[OntologyAnnotation(term='submitter')])
    study.contacts.append(contact)
    publication = Publication(title="Experiments with Elephants", author_list="A.␣
↪Robertson, B. Robertson")
    publication.pubmed_id = "12345678"
    publication.status = OntologyAnnotation(term="published")
    study.publications.append(publication)

    # To create the study graph that corresponds to the contents of the study table␣
↪file (the s_*.txt file), we need
    # to create a process sequence. To do this we use the Process class and attach it␣
↪to the Study object's
    # 'process_sequence' list instance variable. Each process must be linked with a␣
↪Protocol object that is attached to
    # a Study object's 'protocols' list instance variable. The sample collection␣
↪Process object usually has as input
    # a Source material and as output a Sample material.

    # Here we create one Source material object and attach it to our study.

    source = Source(name='source_material')
    study.materials['sources'].append(source)

    # Then we create three Sample objects, with organism as Homo Sapiens, and attach␣
↪them to the study. We use the utility function
    # batch_create_material() to clone a prototype material object. The function␣
↪automatically appends
    # an index to the material name. In this case, three samples will be created,␣
↪with the names
    # 'sample_material-0', 'sample_material-1' and 'sample_material-2'.

    prototype_sample = Sample(name='sample_material', derives_from=[source])
    ncbitaxon = OntologySource(name='NCBITaxon', description="NCBI Taxonomy")
    characteristic_organism = Characteristic(category=OntologyAnnotation(term=
↪"Organism"),
                                             value=OntologyAnnotation(term="Homo Sapiens",␣
↪term_source=ncbitaxon,
                                                                      term_accession="http://
↪purl.bioontology.org/ontology/NCBITAXON/9606"))
    prototype_sample.characteristics.append(characteristic_organism)

    study.materials['samples'] = batch_create_materials(prototype_sample, n=3)  #␣
↪creates a batch of 3 samples
```

```
    # Now we create a single Protocol object that represents our sample collection␣
↪protocol, and attach it to the
    # study object. Protocols must be declared before we describe Processes, as a␣
↪processing event of some sort
    # must execute some defined protocol. In the case of the class model, Protocols␣
↪should therefore be declared
    # before Processes in order for the Process to be linked to one.

    sample_collection_protocol = Protocol(name="sample collection",
                                          protocol_type=OntologyAnnotation(term=
↪"sample collection"))
    study.protocols.append(sample_collection_protocol)
    sample_collection_process = Process(executes_protocol=sample_collection_protocol)

    # Next, we link our materials to the Process. In this particular case, we are␣
↪describing a sample collection
    # process that takes one source material, and produces three different samples.
    #
    # (source_material)->(sample collection)->[(sample_material-0), (sample_material-
↪1), (sample_material-2)]

    for src in study.materials['sources']:
        sample_collection_process.inputs.append(src)
    for sam in study.materials['samples']:
        sample_collection_process.outputs.append(sam)

    # Finally, attach the finished Process object to the study process_sequence. This␣
↪can be done many times to
    # describe multiple sample collection events.

    study.process_sequence.append(sample_collection_process)

    # Next, we build n Assay object and attach two protocols, extraction and␣
↪sequencing.

    assay = Assay(filename="a_assay.txt")
    extraction_protocol = Protocol(name='extraction', protocol_
↪type=OntologyAnnotation(term="material extraction"))
    study.protocols.append(extraction_protocol)
    sequencing_protocol = Protocol(name='sequencing', protocol_
↪type=OntologyAnnotation(term="material sequencing"))
    study.protocols.append(sequencing_protocol)

    # To build out assay graphs, we enumereate the samples from the study-level, and␣
↪for each sample we create an
    # extraction process and a sequencing process. The extraction process takes as␣
↪input a sample material, and produces
    # an extract material. The sequencing process takes the extract material and␣
↪produces a data file. This will
    # produce three graphs, from sample material through to data, as follows:
    #
    # (sample_material-0)->(extraction)->(extract-0)->(sequencing)->(sequenced-data-0)
    # (sample_material-1)->(extraction)->(extract-1)->(sequencing)->(sequenced-data-1)
    # (sample_material-2)->(extraction)->(extract-2)->(sequencing)->(sequenced-data-2)
    #
    # Note that the extraction processes and sequencing processes are distinctly␣
↪separate instances, where the three
```

```
    # graphs are NOT interconnected.

    for i, sample in enumerate(study.materials['samples']):

        # create an extraction process that executes the extraction protocol

        extraction_process = Process(executes_protocol=extraction_protocol)

        # extraction process takes as input a sample, and produces an extract␣
↪material as output

        extraction_process.inputs.append(sample)
        material = Material(name="extract-{}".format(i))
        material.type = "Extract Name"
        extraction_process.outputs.append(material)

        # create a sequencing process that executes the sequencing protocol

        sequencing_process = Process(executes_protocol=sequencing_protocol)
        sequencing_process.name = "assay-name-{}".format(i)
        sequencing_process.inputs.append(extraction_process.outputs[0])

        # Sequencing process usually has an output data file

        datafile = DataFile(filename="sequenced-data-{}".format(i), label="Raw Data␣
↪File", generated_from=[sample])
        sequencing_process.outputs.append(datafile)

        # Ensure Processes are linked forward and backward. plink(from_process, to_
↪process) is a function to set
        # these links for you. It is found in the isatools.model.v1 package

        plink(extraction_process, sequencing_process)

        # make sure the extract, data file, and the processes are attached to the␣
↪assay

        assay.data_files.append(datafile)
        assay.materials['other_material'].append(material)
        assay.process_sequence.append(extraction_process)
        assay.process_sequence.append(sequencing_process)
        assay.measurement_type = OntologyAnnotation(term="gene sequencing")
        assay.technology_type = OntologyAnnotation(term="nucleotide sequencing")

    # attach the assay to the study

    study.assays.append(assay)

    import json
    from isatools.isajson import ISAJSONEncoder

    # To write JSON out, use the ISAJSONEncoder class with the json package and use␣
↪dump() or dumps()
    # Note that the extra parameters sort_keys, indent and separators are to make the␣
↪output more human-readable.

    return json.dumps(investigation, cls=ISAJSONEncoder, sort_keys=True, indent=4,␣
↪separators=(',', ': '))
```

```
if __name__ == '__main__':
    print(create_descriptor())  # print the result to stdout
```

validateISAtab.py

An example program using the ISA-Tab validator to validate one or more ISA-Tab archives.

```
#!/usr/bin/env python

# Inspired by validateSBML.py example from libSBML Python API

from isatools.isatab import validate
import sys
import os


def main(args):
    """usage: validateISAtab.py inputfile1 [inputfile2 ...]
    """
    if len(args) < 1:
        print(main.__doc__)
        sys.exit(1)

    numfiles = 0
    invalid = 0
    skipped = 0

    totalerrors = 0
    totalwarnings = 0

    for i in range(1, len(args)):
        print("----------------------------------------------------------------------------
↪----")
        if not os.path.isfile(args[i]):
            print("Cannot open file {}, skipping".format(args[i]))
            skipped += 1
            numfiles += 1
        else:
            with open(args[i]) as fp:
                report = validate(fp)
                numerrors = len(report['errors'])
                numwarnings = len(report['warnings'])
                if numerrors > 0:
                    invalid += 1
                print("Validator found {} errors and {} warnings in this ISA-Tab␣
↪archive".format(numerrors, numwarnings))
                totalerrors += numerrors
                totalwarnings += numwarnings
                numfiles += 1
    print("----------------------------------------------------------------------------
↪")
    print("Validated {} ISA-Tab archives, {} valid ISA-Tab archives, {} invalid ISA-
↪Tab archives"
          .format(numfiles - skipped, numfiles - invalid - skipped, invalid))
```

```
    print("Found {} errors and {} warnings in across all ISA-Tab archives".
↪format(totalerrors, totalwarnings))

    if invalid > 0:
        sys.exit(1)

if __name__ == '__main__':
    main(sys.argv)
```

validateISAjson.py

An example program using the ISA-JSON validator to validate one or more ISA-JSON files.

```
#!/usr/bin/env python

# Inspired by validateSBML.py example from libSBML Python API

from isatools.isajson import validate
import sys
import os


def main(args):
    """usage: validateISAjson.py inputfile1 [inputfile2 ...]
    """
    if len(args) < 1:
        print(main.__doc__)
        sys.exit(1)

    numfiles = 0
    invalid = 0
    skipped = 0

    totalerrors = 0
    totalwarnings = 0

    for i in range(1, len(args)):
        print("-------------------------------------------------------------------------
↪----")
        if not os.path.isfile(args[i]):
            print("Cannot open file {}, skipping".format(args[i]))
            skipped += 1
            numfiles += 1
        else:
            with open(args[i]) as fp:
                report = validate(fp)
                numerrors = len(report['errors'])
                numwarnings = len(report['warnings'])
                if numerrors > 0:
                    invalid += 1
                print("Validator found {} errors and {} warnings in this ISA-JSON file
↪".format(numerrors, numwarnings))
                totalerrors += numerrors
                totalwarnings += numwarnings
                numfiles += 1
```

```
    print("-------------------------------------------------------------------------
↪")
    print("Validated {} ISA-JSONs, {} valid ISA-JSONs, {} invalid ISA-JSONs"
          .format(numfiles - skipped, numfiles - invalid - skipped, invalid))
    print("Found {} errors and {} warnings in across all ISA-JSONs".
↪format(totalerrors, totalwarnings))

    if invalid > 0:
        sys.exit(1)

if __name__ == '__main__':
    main(sys.argv)
```

ISA Conversions

The ISA API includes a set of functions to allow you to convert between ISA formats, as well as between ISA formats and other formats such as SRA. These converters can be found in the `isatools.convert` package.

Converting from ISA-Tab to ISA JSON

To convert from a directory `./tabdir/` containing valid ISA-Tab files (e.g. `i_investigation.txt`, `s_...txt` and `a_...txt` files):

```
from isatools.convert import isatab2json
isa_json = isatab2json.convert('./tabdir/')
```

Hint: The conversions by default run the ISA validator to check for correctness of the input content. To skip the validation step, set the `validate_first` parameter to `False` by doing something like `converter.convert('./my/path/', validate_first=False)`.

Hint: The conversions by default use a legacy ISA-Tab parser, which has now been replaced with a faster version. To specify using the new parser, set the `use_new_parser` parameter to `True` by doing something like `isatab2json.convert('./my/path/', use_new_parser=True)`.

Converting from ISA JSON to ISA-Tab

To convert from a ISA JSON file `isa.json` directory to write out ISA-Tab files to a target directory `./outdir/`:

```
from isatools.convert import json2isatab
with open('isa.json')) as file_pointer:
    json2isatab.convert(file_pointer, './outdir/')
```

To turn off pre-conversion validation, use *validate_first=False*. By default it is set to *validate_first=True*.

ISA-SRA Conversions

Converting from ISA-Tab to SRA XML

To convert from a directory `./tabdir/` containing valid ISA-Tab files to a write the SRA XML files to a target directory `./outdir/`, validating against a given configuration in `./isaconfig-default_v2015-07-02/`:

```
from isatools.convert import isatab2sra
sra_settings={
          "sra_broker": "MYORG",
          "sra_center": "MYORG",
          "sra_project": "MYORG",
          "sra_broker_inform_on_status": "support@myorg.org",
          "sra_broker_inform_on_error": "support@myorg.org",
          "sra_broker_contact_name": "Support"
      }
isatab2sra.convert('./tabdir/', './outdir/', sra_settings=sra_settings)
```

This method writes the SRA files out to `./outdir/`.

Note that when subitting SRA XML to ENA, you need to supply broker information as shown above in the `sra_settings` JSON, customised to your own organisation's settings.

Converting from ISA JSON file to SRA XML

To convert from a a ISA JSON file `isa.json` directory to write out SRA XML files to a target directory `./outdir/` :

```
sra_settings={
          "sra_broker": "MYORG",
          "sra_center": "MYORG",
          "sra_project": "MYORG",
          "sra_broker_inform_on_status": "support@myorg.org",
          "sra_broker_inform_on_error": "support@myorg.org",
          "sra_broker_contact_name": "Support"
      }
from isatools.convert import json2sra
json2sra.convert(open('isa.json')), './outdir/', sra_settings=sra_settings)
```

This method writes the SRA files out to `./outdir/`.

Note that when subitting SRA XML to ENA, you need to supply broker information as shown above in the `sra_settings` JSON, customised to your own organisation's settings.

To turn off pre-conversion validation, use `validate_first=False`. By default it is set to `validate_first=True`.

Downloading files stored in Github

The ISA GitHub Adapter class

The GitHub API wrapper/adapter may be useful to retrieve ISA datasets (as JSON or Tabs) or configuration files in XML format. The core class, `IsaGitHubStorageAdapter` can be instantiated with or without authorisation.

Usage without authentication

If authentication is not required to access the required resource, you can directly instantiate an adapter object and use it.

```
from isatools.io.storage_adapter import IsaGitHubStorageAdapter
from zipfile import ZipFile
adapter = IsaGitHubStorageAdapter()
adapter.retrieve('tests/data/BII-I-1', 'test_out_dir', owner='ISA-tools', repository=
↪'isa-api')
# retrieving a directory (containg either an ISA-tab dataset or a set of␣
↪configuration files,
# will return a file-like object containg the zipped content of the directory.
buf = adapter.retrieve('tests/data/BII-I-1', destination='test_out_dir', owner='ISA-
↪tools',
                       repository='isa-api')
# Default owner is "ISA-tools' and default repo is 'isa-api' so they can actually be␣
↪omitted.
# Default destination directory is 'isa-target'
zip_file = ZipFile(buf)
# get the list of the files retrieved from the directory
zip_file.namelist()
# an ISA JSON dataset is returned as a stardard JSON object
json_obj = adapter.retrieve('isatools/sampledata/BII-I-1.json', destination='test_out_
↪dir',
                            owner='ISA-tools', repository='isa-api', validate_
↪json=True)
# set write_to_file to False to avoid saving the resource to disk
json_obj = adapter.retrieve('isatools/sampledata/BII-I-1.json', write_to_file=False,
                            owner='ISA-tools', repository='isa-api', validate_
↪json=True)
# retrieving a single configuration file returns an lxml ElementTree object:
xml_obj = adapter.retrieve('isaconfig-2013222/protein_expression_ge.xml',
                           repository='Configuration-Files')
# get root element for the configuration file
xml_obj.getroot()
```

Usage with authentication

To access as authenticated user, the recommended way is to instantiate the storage adapter in a with statement.

```
with IsaGitHubStorageAdapter(username='yourusername', password='yourpw',
                             note='test_api') as adapter:
    adapter.is_authenticated # true
    # do stuff...
```

Otherwise you must explicitly call the `close()` method to delete the current authorisation from the GitHub server

```
adapter = IsaGitHubStorageAdapter(username='yourusername', password='youpw', note=
↪'test_api')
adapter.is_authenticated # True
# do stuff...
adapter.close()
```

Validating ISA-Tab and ISA JSON

Using the ISA API you can validate ISA-Tab and ISA JSON files. The ISA-Tab validation utilises the legacy Java validator, so you must have Java 1.6 or later installed on your host machine. The new validators are in pure Python 3+.

Validating ISA-Tab (legacy Java validator)

To validate ISA-Tab files in a given directory `./tabdir/` against a given configuration found in a directory `./isaconfig-default_v2015-07-02/`, do something like the following:

```
from isatools import isatab
isatab.validate('./tabdir/', './isaconfig-default_v2015-07-02/')
```

to run the legacy Java ISA-Tab validator.

Validating ISA-Tab (native Python implementation)

From v0.2+ of the ISA API, we have started implementing a replacement validator written in Python. To use this one, do something like:

```
from isatools import isatab
my_json_report = isatab.validate2(open('i_investigation.txt'), './isaconfig-default_
↪v2015-07-02/')
```

making sure to *point to the investigation file* of your ISA-Tab, and again providing the XML configurations. The validator will then read the location of your study and assay table files from the investigation file in order to validate those.

Take care to note that function is called `validate2()` and not `validate()`.

This new ISA-Tab validator has been tested against the sample data sets BII-I-1, BII-S-3 and BII-S-7, that are found in the `isatools` package.

The validator will return a JSON-formatted report of warnings and errors.

Validating ISA JSON

To validate an ISA JSON file against the ISA JSON version 1.0 specification you can use our new validator from v0.2, by doing this by doing something like:

```
from isatools import isajson
my_json_report = isajson.validate(open('isa.json'))
```

The rules we check for in the new validators are documented in this working document in Google spreadsheets. Please be aware as this is a working document, some of these rules may be amended as we get more feedback and evolve the ISA API code.

This ISA JSON validator has been tested against a range of dummy test data found in `isatools` tests package.

The validator will return a JSON-formatted report of warnings and errors.

Batch validation of ISA-Tab and ISA-JSON

To validate a batch of ISA-Tabs or ISA-JSONs, you can use the `batch_validate()` function.

To validate a batch of ISA-Tabs, you can do something like:

```
from isatools import isatab
my_tabs = [
    '/path/to/study1/',
    '/path/to/study2/'
]
my_json_report = isatab.batch_validate(my_tabs, '/path/to/report.txt')
```

To validate a batch of ISA-JSONs, you can do something like

```
from isatools import isajson
my_jsons = [
    '/path/to/study1.json',
    '/path/to/study2.json'
]
my_json_report = isajson.batch_validate(my_jsons, '/path/to/report.txt')
```

In both cases, the batch validation will return a JSON-formatted report of warnings and errors.

Reformatting JSON reports

The JSON reports produced by the validators can be reformatted using a function found in the `isatools.utils` module.

For example, to write out the report as a CSV textfile to `report.txt`, you can do something like:

```
from isatools import utils
with open('report.txt', 'w') as report_file:
    report_file.write(utils.format_report_csv(my_json_report))
```

Importing data into ISA formats

We have provided a number of modules that allow you to import data into ISA formats from well-known databases or services.

Importing from the MetaboLights database, to ISA

To import an MetaboLights study from the MetaboLights as ISA-Tab files, provide an MetaboLights accession number:

```
from isatools.io import mtbls as MTBLS
tmp_dir = MTBLS.get('MTBLS1')
```

This method downloads the ISA-Tab files for a study, and returns a string path to a temporary directory containing the ISA-Tab files.

To import an MetaboLights study from the MetaboLights as ISA JSON files, provide an MetaboLights accession number:

```
from isatools.io import mtbls as MTBLS
isa_json = MTBLS.getj('MTBLS1')
```

This method gets the study and returns the ISA content as ISA JSON.

You can also do simple queries on MetaboLights studies to retrieve samples and related data files, based on factor selection:

```
from isatools.io import mtbls as MTBLS
MTBLS.get_factor_names('MTBLS1')
# response:
# {'Gender', 'Age'}
MTBLS.get_factor_values('MTBLS1', 'Gender')
# response:
# {'Male', 'Female'}
query = {
      "Gender": "Male"
   }
samples_and_files = MTBLS.get_data_files('MTBLS1', factor_query=query)
# response:
#  [
#     {
#        'sample': 'ADG10003u_007'},
#        'data_files': ['ADG10003u_007.zip'],
#        'query_used': {'Gender': 'Male'}
#     }, ...
#  ]
```

Importing SRA from the European Nucleotide Archive, to ISA-Tab

Notice: this method depends on SAXON XSLT Processor

To import an SRA study from the European Nucleotide Archive (ENA) as ISA-Tab files, provide an ENA accession number and your path to the SAXON JAR file:

```
from isatools.convert import sra2isatab
sra2isatab.sra_to_isatab_batch_convert('BN000001', 'your/path/to/saxon9.jar')
```

This method returns the ISA-Tab files as a byte stream (`io.BytesIO`).

Importing from MetabolomicsWorkbench, to ISA-Tab

To import a study from the Metabolomics Workbench as ISA-Tab files, provide an accession number and your local path to write your files to:

```
from isatools.convert.mw2isa import mw2isa_convert
success, study_id, validate = mw2isa_convert(studyid="ST000367", outputdir='tmp/', dl_
↪option="no", validate_option="yes")
#  If success == True, download and conversion ran OK. If validate == True, the ISA-
↪Tabs generated passed validation
```

See `isa-api/isatools/convert/mw2isa.py`

Importing from Biocrates, to ISA-Tab

Notice: this method depends on SAXON XSLT Processor

See `isa-api/isatools/convert/biocrates2isatab.py`

Importing mzML to ISA-Tab

To import metadata from mzML mass spectrometry files, the ISA API integrates with the `mzml2isa` tool from https://github.com/ISA-tools/mzml2isa and can be run as follows:

```
from isatools.convert import mzml2isa
mzml2isa.convert('your/path/to/mzml/files/', 'tmp/', "My Study ID")
```

Importing SampleTab to ISA

To import metadata from SampleTab files (e.g. from EBI BioSamples database), you can do the following to import a SampleTab to ISA-Tab:

```
from isatools.convert import sampletab2isatab
with open('your/path/to/sampletab.txt', 'r') as input_sampletab:
    sampletab2isatab.convert(input_sampletab, 'tmp/')
```

To import a SampleTab to ISA JSON, you can do:

```
from isatools.convert import sampletab2json
with open('your/path/to/sampletab.txt', 'r') as input_sampletab:
    with open('your/path/to/myjson.json', 'w') as output_json:'
        sampletab2json.convert(input_sampletab, output_json)
```

You can also load SampleTab content directly into ISA Python objects:

```
from isatools import sampletab
with open('your/path/to/sampletab.txt', 'r') as input_sampletab:
    ISA = sampletab.load(input_sampletab)
```

Exporting data from ISA formats

We have provided a number of modules that allow you to export data from ISA formats to formats for well-known databases or services.

Exporting ISA content to SampleTab

To export metadata from SampleTab files (e.g. for EBI BioSamples database), you can do the following to export a ISA-Tab to SampleTab:

```
from isatools.convert import isatab2sampletab
with open('your/path/to/i_investigation.txt', 'r') as input_investigation_file:
    with open('your/path/to/sampletab.txt', 'w') as output_sampletab_file:
        isatab2sampletab.convert(input_investigation_file, output_sampletab_file)
```

To export an ISA JSON file to SampleTab, you can do:

```
from isatools.convert import isatab2sampletab
with open('your/path/to/i_investigation.txt', 'r') as input_investigation_file:
    with open('your/path/to/sampletab.txt', 'w') as output_sampletab_file:
        isatab2sampletab.convert(input_investigation_file, output_sampletab_file)
```

You can also dump SampleTab content directly from ISA Python objects:

```
from isatools import sampletab
with open('your/path/to/sampletab.txt', 'w') as output_sampletab:
    # Note: ISA would be a previously loaded or constructed root Investigation object
    sampletab.dump(ISA, output_sampletab)
```

Known issues

isatools v0.8 package

- Issues #153 is still outstanding, as per below; new issue #208 (ISA-Tab validation issue)
- SRA/ENA importer and Biocrates importer relies on XSLT2 processing only available with SAXON and requires .jar file to run

isatools v0.7 package

- Issues #101, #153 are still outstanding, as per below
- SRA/ENA importer and Biocrates importer relies on XSLT2 processing only available with SAXON and requires .jar file to run

isatools v0.6 package

- Issues #146, #101, #153 are still outstanding, as per below
- SRA/ENA importer and Biocrates importer relies on XSLT2 processing only available with SAXON and requires .jar file to run
- We are aware that there may be some performance issues when loading and writing ISA-Tab documents with several thousand samples. This has been worked on and there is ongoing efforts to improve on the current performance of ISA-Tab I/O

isatools v0.5 package

- All issues inherited from v0.4 (see below)
- Currently only Python 3.4 and 3.5 is supported. Python 2.7 support is present in the `py2` source branch on Github.

isatools v0.4 package

- For certain of ISA-Tab table files, the ISA-Tab parser cannot disambiguate between process instances where a `Name` column is required to qualify a `Protocol REF` has been left blank. Utility functions have been written to detect these anomalies and to assist in correcting them, in the `isatools.utils` package. #146 (see detail after bullet points)
- When converting to ISA JSON in using `UUID` or `counter` Identifier Types, some elements are not detected, such as `Array_Design_REF` #101
- The ISA-Tab parser does not support reading `Protein Assignment File`, `Peptide Assignment File`, `Post Translational Modification Assignment File` columns, and therefore the `isatab2*` converters also do not support these #174
- The SRA/ENA importer in `sra2isatab` relies on XSLT2 processing functionality only available with SAXON, so you must provide the JAR file yourself to use this
- `sra2isatab` converter does not support SRA pools #153
- The legacy functionality (marked in the documentation) relies on Java 1.6

To check for possible erroneous pooling events in an ISA-Tab archive, you can try something like:

```
>>> from isatools import utils
>>> utils.detect_isatab_process_pooling('tests/data/tab/MTBLS1/')
INFO: Converting ISA-Tab to ISA JSON...
INFO: Converting ISAtab to ISAjson for tests/data/tab/MTBLS1/
INFO: ... conversion finished.
Checking s_MTBLS1.txt
Checking a_mtbls1_metabolite_profiling_NMR_spectroscopy.txt
Possible process pooling detected on:  #process/Extraction1
Possible process pooling detected on:  #process/ADG_normalized_data.xlsx
[{'a_mtbls1_metabolite_profiling_NMR_spectroscopy.txt': ['#process/Extraction1', '
↪#process/ADG_normalized_data.xlsx']}]
>>>
```

In this case, `#process/Extraction1` is the pooling that we did not expect. This is a pooling on a single `Extraction`. From manual inspection of the ISA-Tab file `a_mtbls1_metabolite_profiling_NMR_spectroscopy.txt` we can

then confirm that values are entirely missing from `Extract Name`, causing the parser to think the experimental graph converges on one process node. To rectify this, individual values should be put into this `Name` column. We can try

fix erroneous pooling by filling out an empty `Name` column with a corresponding `Protocol REF` by doing the following:

```
>>> utils.insert_distinct_parameter(open('tests/data/tab/MTBLS1/a_mtbls1_metabolite_
↪profiling_NMR_spectroscopy.txt', 'r+'), 'Extraction')
Are you sure you want to add a column of hash values in Extract Name? Y/(N)
>? Y
```

If successful, this will fill out the empty column with 8 character-long UUIDs (e.g. 4078cb03).

Please be aware that these utility functions `detect_isatab_process_pooling()` and `insert_distinct_parameter()` are there to help you manually fix your ISA-Tabs, not to automatically fix them for you. We wil address this issue in more depth in following releases.

isatools v0.3 package

- `required` constraints on JSON schemas causes validation failure for `@id` objects, meaning some constraints using JSON schemas cannot be used for validation #108
- Chained processes (i.e. a process followed by another process without any intermediate inputs and outputs, in ISAtab a `Protocol REF` columns followed by another `Protocol REF` columns without any materials in between) are not currently supported. It is not recommended to attempt to use such patterns with this version of the `isatools` package #111
- When converting to ISA JSON in using `UUID` or `counter` Identifier Types, some elements are not detected, such as `Array_Design_REF` #101
- The SRA/ENA importer in `sra2isatab` relies on XSLT2 processing functionality only available with SAXON, so you must provide the JAR file yourself to use this
- The legacy functionality (marked in the documentation) relies on Java 1.6

isatools v0.2 package

- `required` constraints on JSON schemas causes validation failure for `@id` objects, meaning some constraints using JSON schemas cannot be used for validation #108
- When converting to ISA JSON in using `UUID` or `counter` Identifier Types, some elements are not detected, such as `Array_Design_REF` #101
- `Protocol REF` columns must be present in order for the ISA-Tab to JSON conversion to pick up processes in the process sequences #111
- Characteristics and Factor Values declared in assay tables in ISAtab are associated to Sample objects only. This means that when writing from Python objects, or converting from ISA JSON, to ISAtab these columns appear at the study table.
- Chained processes (i.e. a process followed by another process without any intermediate inputs and outputs, in ISAtab a `Protocol REF` columns followed by another `Protocol REF` columns without any materials in between) are not currently supported. It is not recommended to attempt to use such patterns with this version of the `isatools` package #111
- For experimental graph patterns to work, should follow relatively simple patterns. e.g. Straight Sample -> ... -> Materials -> ... -> Data paths (per assay), or simple splitting and pooling. See test package code for examples of what works.
- No ISA JSON configurations have been included that correspond with the following default XML configurations: `clinical_chemistry.xml` and most are as yet untested.

isatools v0.1 package

- Characteristics and Factor Values declared in assay tables in ISAtab are associated to Sample objects only. This means that when writing from Python objects, or converting from ISA JSON, to ISAtab these columns appear at the study table.
- Chained processes (i.e. a process followed by another process without any intermediate inputs and outputs, in ISAtab a `Protocol REF` columns followed by another `Protocol REF` columns without any materials in between) are not currently supported. It is not recommended to attempt to use such patterns with this version of the `isatools` package #111

- For experimental graph patterns to work, should follow relatively simple patterns. e.g. Straight Sample -> ... -> Materials -> ... -> Data paths (per assay), or simple splitting and pooling. See test package code for examples of what works.

For a full up-to-date list of issues, or to report an issue or ask a question, please see the issue tracker.