MSL-IO Documentation

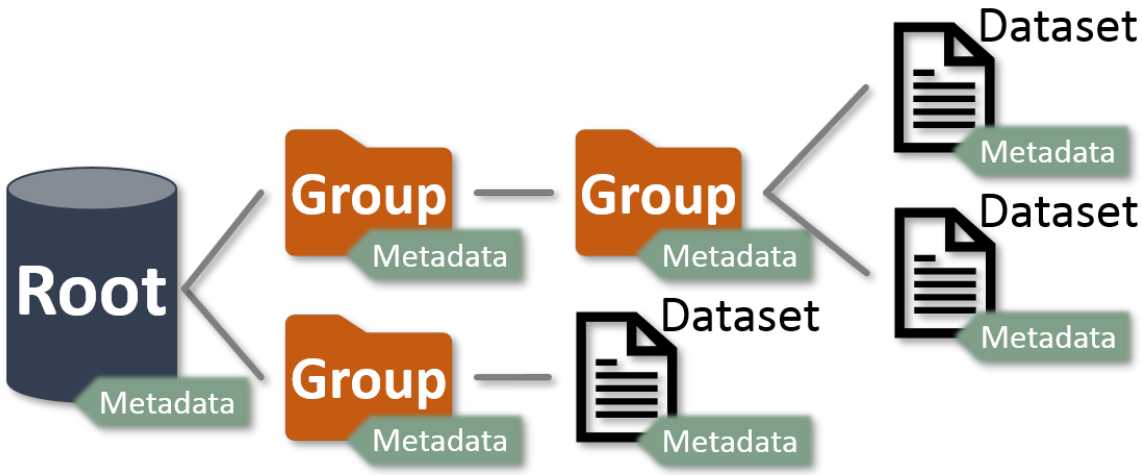
Release 0.2.0.dev0

Measurement Standards Laboratory of New Zealand

Jul 09, 2023

CONTENTS

MSL-IO follows the data model used by HDF5 to read and write data files – where there is a *Root*, *Group*s and *Dataset*s and these objects each have *Metadata* associated with them.



The tree structure is similar to the file-system structure used by operating systems. *Group*s are analogous to the directories (where *Root* is the root *Group*) and *Dataset*s are analogous to the files.

The data files that can be read or created are not restricted to HDF5 files, but any file format that has a *Reader* implemented can be read and data files can be created using any of the *Writers*.

CHAPTER

ONE

GETTING STARTED

- *Write a file*
- *Read a file*
- *Convert a file*
- *Read data in a table*

CHAPTER

TWO

WRITE A FILE

Suppose you want to create a new HDF5 file. We first create an instance of *HDF5Writer*

```
>>> from msl.io import HDF5Writer
>>> h5 = HDF5Writer()
```

then we can add *Metadata* to the *Root*,

```
>>> h5.add_metadata(one=1, two=2)
```

create a *Dataset* in the *Root*,

```
>>> dataset1 = h5.create_dataset('dataset1', data=[1, 2, 3, 4])
```

create a *Group* in the *Root*,

```
>>> my_group = h5.create_group('my_group')
```

and create a *Dataset* in *my_group*

```
>>> dataset2 = my_group.create_dataset('dataset2', data=[[1, 2], [3, 4]],␣
↪three=3)
```

Finally, we write the file

```
>>> h5.write(file='my_file.h5')
```

Note: The file is not created until you call the *write()* or *save()* method.

CHAPTER

THREE

READ A FILE

The *read()* function is available to read a file. Provided that a *Reader* exists to read the file a *Root* object is returned. We will read the file that we created above.

```
>>> from msl.io import read
>>> root = read('my_file.h5')
```

You can print a representation of all *Group*s and *Dataset*s in the *Root* by calling the *tree()* method

```
>>> print(root.tree())
<HDF5Reader 'my_file.h5' (1 groups, 2 datasets, 2 metadata)>
  <Dataset '/dataset1' shape=(4,) dtype='<f8' (0 metadata)>
  <Group '/my_group' (0 groups, 1 datasets, 0 metadata)>
    <Dataset '/my_group/dataset2' shape=(2, 2) dtype='<f8' (1 metadata)>
```

Since the *root* object is a *Group* (which operates like a Python `dict`) you can iterate over the items that are in the file using

```
>>> for name, value in root.items():
...     print('{!r} -- {!r}'.format(name, value))
'/dataset1' -- <Dataset '/dataset1' shape=(4,) dtype='<f8' (0 metadata)>
'/my_group' -- <Group '/my_group' (0 groups, 1 datasets, 0 metadata)>
'/my_group/dataset2' -- <Dataset '/my_group/dataset2' shape=(2, 2) dtype='<f8
↪' (1 metadata)>
```

where *value* will either be a *Group* or a *Dataset*.

You can iterate over the *Group*s that are in the file

```
>>> for group in root.groups():
...     print(group)
<Group '/my_group' (0 groups, 1 datasets, 0 metadata)>
```

or iterate over the *Dataset*s

```
>>> for dataset in root.datasets():
...     print(repr(dataset))
<Dataset '/dataset1' shape=(4,) dtype='<f8' (0 metadata)>
<Dataset '/my_group/dataset2' shape=(2, 2) dtype='<f8' (1 metadata)>
```

You can access the *Metadata* of any object through the *metadata* attribute

```
>>> root.metadata
<Metadata '/' {'one': 1, 'two': 2}>
```

You can access values of the *Metadata* as attributes

```
>>> root.metadata.one
1
>>> dataset2.metadata.three
3
```

or as keys

```
>>> root.metadata['two']
2
>>> dataset2.metadata['three']
3
```

When *root* is returned it is accessed in read-only mode

```
>>> root.read_only
True
>>> for name, value in root.items():
...     print('is {!r} in read-only mode? {}'.format(name, value.read_only))
is '/dataset1' in read-only mode? True
is '/my_group' in read-only mode? True
is '/my_group/dataset2' in read-only mode? True
```

If you want to edit the `Metadata` for *root*, or modify any `Group`s or `Dataset`s in *root*, then you must first set the object to be editable. Setting the read-only mode of *root* propagates that mode to all items within *root*. For example,

```
>>> root.read_only = False
```

will make *root* and all *Group*s and all *Dataset*s within *root* to be editable

```
>>> root.read_only
False
>>> for name, value in root.items():
...     print('is {!r} in read-only mode? {}'.format(name, value.read_only))
is '/dataset1' in read-only mode? False
is '/my_group' in read-only mode? False
is '/my_group/dataset2' in read-only mode? False
```

You can make only a specific object (and it's descendants) editable as well. You can make *my_group* and *dataset2* to be in read-only mode by the following (recall that *root* behaves like a Python `dict`)

```
>>> root['my_group'].read_only = True
```

and this will keep *root* and *dataset1* in editable mode, but change *my_group* and *dataset2* to be in read-only mode

```
>>> root.read_only
False
>>> for name, value in root.items():
...     print('is {!r} in read-only mode? {}'.format(name, value.read_only))
is '/dataset1' in read-only mode? False
is '/my_group' in read-only mode? True
is '/my_group/dataset2' in read-only mode? True
```

You can access the *Group*s and *Dataset*s as keys or as class attributes

```
>>> root['my_group']['dataset2'].shape
(2, 2)
>>> root.my_group.dataset2.shape
(2, 2)
```

See *Accessing Keys as Class Attributes* for more information.

CHAPTER

FOUR

CONVERT A FILE

You can convert between file formats using any of the *Writers*. Suppose you had an HDF5 file and you wanted to convert it to the JSON format

```
>>> from msl.io import JSONWriter
>>> h5 = read('my_file.h5')
>>> writer = JSONWriter('my_file.json')
>>> writer.write(root=h5)
```

CHAPTER

FIVE

READ DATA IN A TABLE

The *read_table()* function is available to read a table from a file.

A *table* has the following properties:

1. The first row is a header.
2. All rows have the same number of columns.
3. All data values in a column have the same data type.

The returned object is a *Dataset* with the header provided as metadata.

Suppose a file called *my_table.csv* contains the following information

You can read this file and interact with the data using the following

```
>>> from msl.io import read_table
>>> csv = read_table('my_table.csv')
>>> csv
<Dataset 'my_table.csv' shape=(3, 3) dtype='<f8' (1 metadata)>
>>> csv.metadata
<Metadata 'my_table.csv' {'header': array(['x', 'y', 'z'], dtype='<U1')}>
>>> csv.data
array([[1., 2., 3.],
       [4., 5., 6.],
       [7., 8., 9.]])
>>> csv.max()
9.0
```

You can read a table from a text-based file or from an Excel spreadsheet.

CHAPTER

SIX

CONTENTS

6.1 Install MSL-IO

To install **MSL-IO** run

```
pip install msl-io
```

Alternatively, using the MSL Package Manager run

```
msl install io
```

6.1.1 Dependencies

- Python 3.8+
- numpy
- xlrd (bundled with **MSL-IO**)

6.1.2 Optional Dependencies

The following packages are not automatically installed when **MSL-IO** is installed but may be required to read some data files.

- h5py
- google-api-python-client
- google-auth-httplib2
- google-auth-oauthlib

To include h5py when installing **MSL-IO** run

```
msl install io[h5py]
```

To include the Google-API packages when installing **MSL-IO** run

```
msl install io[google]
```

6.2 Group

A *Group* is analogous to a directory for an operating system. A *Group* can contain any number of sub-*Group*s (i.e., sub-directories) and it can contain any number of *Dataset*s. It uses a naming convention analogous to UNIX file systems where every sub-directory is separated from its parent directory by the `'/'` character.

From a Python perspective, a *Group* operates like a `dict`. The *keys* are the names of *Group* members, and the *values* are the members themselves (*Group* or `Dataset` objects).

```
>>> print(root.tree())
 <JSONWriter 'example.json' (3 groups, 1 datasets, 0 metadata)>
   <Group '/a' (2 groups, 1 datasets, 0 metadata)>
     <Group '/a/b' (1 groups, 1 datasets, 0 metadata)>
       <Group '/a/b/c' (0 groups, 1 datasets, 0 metadata)>
         <Dataset '/a/b/c/dset' shape=(100,) dtype='<f8' (0 metadata)>
```

A *Group* can be in read-only mode, but can also be set to editable mode

```
>>> b.create_dataset('dset_b', data=[1, 2, 3, 4])
Traceback (most recent call last):
  ...
ValueError: Cannot modify <Group '/a/b' (1 groups, 1 datasets, 0 metadata)>.␣
↪It is accessed in read-only mode.
>>> b.read_only = False
>>> b.create_dataset('dset_b', data=[1, 2, 3, 4])
<Dataset '/a/b/dset_b' shape=(4,) dtype='<f8' (0 metadata)>
```

The *keys* of a *Group* can also be accessed as class attributes

```
>>> root['a']['b']['c']['dset']
<Dataset '/a/b/c/dset' shape=(100,) dtype='<f8' (0 metadata)>
>>> root.a.b.c.dset
<Dataset '/a/b/c/dset' shape=(100,) dtype='<f8' (0 metadata)>
```

See *Accessing Keys as Class Attributes* for more information.

You can navigate through the tree by considering a *Group* to be an ancestor or descendant of other *Group*s

```
>>> for ancestor in c.ancestors():
...     print(ancestor)
 <Group '/a/b' (1 groups, 2 datasets, 0 metadata)>
 <Group '/a' (2 groups, 2 datasets, 0 metadata)>
 <JSONWriter 'example.json' (3 groups, 2 datasets, 0 metadata)>
>>> for descendant in b.descendants():
...     print(descendant)
 <Group '/a/b/c' (0 groups, 1 datasets, 0 metadata)>
```

6.3 Dataset

A *Dataset* is analogous to a file for an operating system and it is contained within a *Group*.

A *Dataset* is essentially a numpy.ndarray with *Metadata* and it can be accessed in read-only mode.

Since a *Dataset* can be thought of as an numpy.ndarray the attributes of an numpy.ndarray are also valid for a *Dataset*. For example, suppose *my_dataset* is a *Dataset*

```
>>> my_dataset
<Dataset '/my_dataset' shape=(5,) dtype='|V16' (2 metadata)>
>>> my_dataset.data
array([(0.23, 1.27), (1.86, 2.74), (3.44, 2.91), (5.91, 1.83), (8.73, 0.74)],
      dtype=[('x', '<f8'), ('y', '<f8')])
```

You can get the numpy.ndarray.shape using

```
>>> my_dataset.shape
(5,)
```

or convert the data in the *Dataset* to a Python list, using numpy.ndarray.tolist()

```
>>> my_dataset.tolist()
[(0.23, 1.27), (1.86, 2.74), (3.44, 2.91), (5.91, 1.83), (8.73, 0.74)]
```

To access the *Metadata* of a *Dataset*, you call the *metadata* attribute

```
>>> my_dataset.metadata
<Metadata '/my_dataset' {'temperature': 20.13, 'humidity': 45.31}>
```

You can access values of the *Metadata* as attributes

```
>>> my_dataset.metadata.temperature
20.13
```

or as keys

```
>>> my_dataset.metadata['humidity']
45.31
```

Depending on the numpy.dtype that was used to create the underlying numpy.ndarray for the *Dataset* the field names can also be accessed as field attributes. For example, you can access the fields in *my_dataset* as keys

```
>>> my_dataset['x']
array([0.23, 1.86, 3.44, 5.91, 8.73])
```

or as attributes

```
>>> my_dataset.x
array([0.23, 1.86, 3.44, 5.91, 8.73])
```

Note that the returned object is a numpy.ndarray and therefore does not contain any *Metadata*.

See *Accessing Keys as Class Attributes* for more information.

You can also chain multiple attribute calls together. For example, to get the maximum *x* value in *my_dataset* you can use

```
>>> my_dataset.x.max()
8.73
```

6.3.1 Slicing and Indexing

Slicing and indexing a *Dataset* is a valid operation, but returns a numpy.ndarray which does not contain any *Metadata*.

Consider *my_dataset* from above. One can slice it

```
>>> my_dataset[::2]
array([(0.23, 1.27), (3.44, 2.91), (8.73, 0.74)],
      dtype=[('x', '<f8'), ('y', '<f8')])
```

or index it

```
>>> my_dataset[2]
(3.44, 2.91)
```

Since a numpy.ndarray is returned, you are responsible for keeping track of the *Metadata* in slicing and indexing operations. For example,

```
>>> my_subset = root.create_dataset('my_subset', data=my_dataset[::2], **my_
↪dataset.metadata)
>>> my_subset
<Dataset '/my_subset' shape=(3,) dtype='|V16' (2 metadata)>
>>> my_subset.data
array([(0.23, 1.27), (3.44, 2.91), (8.73, 0.74)],
      dtype=[('x', '<f8'), ('y', '<f8')])
>>> my_subset.metadata
<Metadata '/my_subset' {'temperature': 20.13, 'humidity': 45.31}>
```

6.3.2 Arithmetic Operations

Arithmetic operations are valid with a *Dataset*, however, the returned object will be a numpy.ndarray and therefore all *Metadata* of the *Dataset*s that are involved in the operation are not included in the returned object.

For example, suppose you have two *Dataset*s that contain the following information

```
>>> dset1
<Dataset '/dset1' shape=(3,) dtype='<f8' (1 metadata)>
>>> dset1.data
array([1., 2., 3.])
>>> dset1.metadata
<Metadata '/dset1' {'temperature': 20.3}>
```

(continues on next page)

(continued from previous page)

```
>>> dset2
<Dataset '/dset2' shape=(3,) dtype='<f8' (1 metadata)>
>>> dset2.data
array([4., 5., 6.])
>>> dset2.metadata
<Metadata '/dset2' {'temperature': 21.7}>
```

You can directly add the *Dataset*s, but the *temperature* values in *Metadata* are not included in the returned object

```
>>> dset3 = dset1 + dset2
>>> dset3
array([5., 7., 9.])
>>> dset3.metadata
Traceback (most recent call last):
  File "<input>", line 1, in <module>
AttributeError: 'numpy.ndarray' object has no attribute 'metadata'
```

You are responsible for keeping track of the *Metadata* in arithmetic operations, for example,

```
>>> temperatures = {'t1': dset1.metadata.temperature, 't2': dset2.metadata.
↪temperature}
>>> dset3 = root.create_dataset('dset3', data=dset1+dset2,␣
↪temperatures=temperatures)
>>> dset3
<Dataset '/dset3' shape=(3,) dtype='<f8' (1 metadata)>
>>> dset3.data
array([5., 7., 9.])
>>> dset3.metadata
<Metadata '/dset3' {'temperatures': {'t1': 20.3, 't2': 21.7}}>
```

6.3.3 A Dataset for Logging Records

The *DatasetLogging* class is a custom *Dataset* that is also a `Handler` which automatically appends `logging` records to the *Dataset*. See *create_dataset_logging()* for more details.

When a file is *read()* it will load an object that was once a *DatasetLogging* as a *Dataset*. If you want to convert the *Dataset* to be a *DatasetLogging* object, so that `logging` records are once again appended to it, then call the *require_dataset_logging()* method with the *name* argument equal to the value of *name* for the *Dataset*.

6.4 Metadata

All *Group* and *Dataset* objects contain *Metadata*. A *Metadata* object is a `dict` that can be made read only and allows for accessing the *keys* of the `dict` as class attributes (see *Accessing Keys as Class Attributes* for more information).

For example, suppose that a file is read with the *Root Group* having the following *Metadata*

```
>>> root.metadata
<Metadata '/' {'voltage': 1.2, 'voltage_unit': 'V'}>
```

A value can be accessed by specifying a *key*

```
>>> root.metadata['voltage']
1.2
```

or as a class attribute

```
>>> root.metadata.voltage
1.2
```

When a file is read, the *Root* object is returned in read-only mode so you cannot modify the metadata

```
>>> root.metadata.voltage = 7.64
Traceback (most recent call last):
  ...
ValueError: Cannot modify <Metadata '/' {'voltage': 1.2, 'voltage_unit': 'V'}>
↪. It is accessed in read-only mode.
```

However, you can allow *root* to be modified by setting the *read_only* property to be `False`

```
>>> root.metadata.read_only = False
>>> root.metadata.voltage = 7.64
>>> root.add_metadata(current=10.3, current_unit='mA')
>>> root.metadata
<Metadata '/' {'voltage': 7.64, 'voltage_unit': 'V', 'current': 10.3,
↪'current_unit': 'mA'}>
```

6.5 Readers

The following *Reader*s are available:

6.5.1 msl.io.readers.detector_responsivity_system module

Reader for the Detector Responsivity System from Light Standards at MSL.

class msl.io.readers.detector_responsivity_system.**DRSReader**(*file*)

Bases: *Reader*

Reader for the Detector Responsivity System from Light Standards at MSL.

Parameters
file (path-like or file-like) – The file to read.

static can_read(*file*, ***kwargs*)

Checks if the first line starts with DRS and ends with Shindo.

read(***kwargs*)

Reads the *.DAT* and corresponding *.LOG* file.

Parameters
****kwargs** – All key-value pairs are ignored.

6.5.2 msl.io.readers.hdf5 module

Reader for the HDF5 file format.

Attention: This Reader loads the entire HDF5 file in memory. If you need to use any of the more advanced features of an HDF5 file, it is best to directly load the file using H5py.

class msl.io.readers.hdf5.**HDF5Reader**(*file*)

Bases: *Reader*

Reader for the HDF5 file format.

Parameters
file (path-like or file-like) – The file to read.

static can_read(*file*, ***kwargs*)

The HDF5 file format has a standard signature.

The first 8 bytes are \x89HDF\r\n\x1a\n.

read(***kwargs*)

Reads the HDF5 file.

Parameters
****kwargs** – All key-value pairs are passed to File.

6.5.3 msl.io.readers.json_ module

Read a file that was created by *JSONWriter*.

class msl.io.readers.json_.**JSONReader**(*file*)

Bases: *Reader*

Read a file that was created by *JSONWriter*.

Parameters
file (path-like or file-like) – The file to read.

static can_read(*file*, ***kwargs*)

Checks if the text `MSL JSONWriter` is in the first line of the file.

read(***kwargs*)

Read the file that was created by *JSONWriter*

If a *Metadata* *key* has a *value* that is a `list` then the list is converted to an `ndarray` with `dtype = object`

Parameters
****kwargs** – Accepts *encoding* and *errors* keyword arguments which are passed to `open()`. The default *encoding* value is `'utf-8'` and the default *errors* value is `'strict'`. All additional keyword arguments are passed to json.loads.

6.5.4 Create a New Reader

When adding a new *Reader* class to the repository the following steps should be performed. Please follow the style guide.

Note: If you do not want to upload the new *Reader* class to the repository then you only need to write the code found in Step 2 to use your *Reader* in your own program. Once you import your module in your code your *Reader* will be available from the *read()* function.

---

1. Create a fork of the repository.
2. Create a new *Reader* by following this template and save it to the msl/io/readers/ directory.

```
# import the necessary MSL-IO objects
from msl.io import register, Reader

# register your Reader so that Python knows that your Reader␣
↪exists
@register
class AnExampleReader(Reader):
    """Name your class to be whatever you want, i.e., change␣
↪AnExampleReader"""

    @staticmethod
    def can_read(file, **kwargs):
        """This method answers the following question:
```

(continues on next page)

(continued from previous page)

```
        Given a path-like object (e.g., a string, bytes or os.
↪PathLike object)
        that represents the location of a file or a file-like␣
↪object (e.g., a
        stream, socket or in-memory buffer) can your Reader read␣
↪this file?

        You must perform all the necessary checks that␣
↪*uniquely* answers this
        question. For example, checking that the file extension␣
↪is ".csv" is
        not unique enough.

        The optional kwargs can be passed in via the msl.io.
↪read() method.

        This method must return a boolean: True (can read) or␣
↪False (cannot read)
        """
        return boolean

    def read(self, **kwargs):
        """This method reads the data file(s).

        Your Reader class is a Root object. The optional kwargs␣
↪can be
        passed in via the msl.io.read() method.

        The data file to read is available at self.file

        To add metadata to Root use self.add_metadata()

        To create a Group in Root use self.create_group()

        To create a Dataset in Root use self.create_dataset()

        This method should not return anything.
        """
```

3. Import your Reader in the `msl/io/readers/__init__.py` module.

4. Add an example data file to the tests/samples directory and add a test case to the tests/ directory to make sure that your Reader is returned by calling the *`read()`* function using your example data file as the input and that the information in the returned object is correct. Run the tests using `python setup.py tests` (ideally you would run the tests for all *currently-supported versions* of Python, see also condatests.py).

5. Create a new `msl.io.readers.<name of your module from Step 2>.rst` file in docs/_api. Follow the template that is used for the other `.rst` files in this directory.

6. Add the new Reader, alphabetically, to the `.. toctree::` in docs/readers.rst. Follow the template

that is used for the other Readers.

7. Add yourself to `AUTHORS.rst` and add a note in `CHANGES.rst` that you created this new Reader. These files are located in the root directory of the **MSL-IO** package.
8. Build the documentation running `python setup.py docs` (view the documentation by opening the `docs/_build/html/index.html` file).
9. If running the tests pass and building the documentation show no errors/warnings then create a pull request.

6.6 Writers

The following *Writer*s are available:

6.6.1 msl.io.writers.hdf5 module

Writer for the HDF5 file format.

Attention: requires that the h5py package is installed.

class msl.io.writers.hdf5.**HDF5Writer**(*file=None*, ***metadata*)

Bases: *Writer*

Create a HDF5 writer.

You can use *HDF5Writer* as a context manager. For example,

```
with HDF5Writer('my_file.h5') as root:
    root.create_dataset('dset', data=[1, 2, 3])
```

This will automatically write *root* to the specified file when the with block exits.

Parameters

- **file** (path-like or file-like, optional) – The file to write the data to. Can also be specified in the *write()* method.
- ****metadata** – Key-value pairs that are used as *Metadata* of the *Root*.

write(*file=None*, *root=None*, ***kwargs*)

Write to a HDF5 file.

Parameters

- **file** (path-like or file-like, optional) – The file to write the *root* to. If None then uses the value of *file* that was specified when *HDF5Writer* was instantiated.
- **root** (*Root*, optional) – Write *root* in HDF5 format. If None then write the *Group*s and *Dataset*s in this *HDF5Writer*.
- ****kwargs** – All key-value pairs are passed to File.

6.6.2 msl.io.writers.json_ module

Writer for a JSON file format. The corresponding *Reader* is *JSONReader*.

class msl.io.writers.json_.**JSONWriter**(*file=None*, ***metadata*)

Bases: *Writer*

Create a JSON writer.

You can use *JSONWriter* as a context manager. For example,

```
>>> with JSONWriter('example.json') as root:
...     dset = root.create_dataset('dset', data=[1, 2, 3])
...     root.update_context_kwargs(indent=4)
```

This will automatically write *root* to the specified file using `indent=4` as a keyword argument to the *write()* method when the with block exits.

Parameters

- **file** (path-like or file-like, optional) – The file to write the data to. Can also be specified in the *write()* method.
- ****metadata** – Key-value pairs that are used as *Metadata* of the *Root*.

write(*file=None*, *root=None*, ***kwargs*)

Write to a JSON file.

The first line in the output file contains a description that the file was created by the *JSONWriter*. It begins with a # and contains a version number.

Version 1.0 specifications

- Use the `'dtype'` and `'data'` keys to uniquely identify a JSON object as a *Dataset*.
- If a *Metadata* *key* has a *value* that is a *Metadata* object then the *key* becomes the name of a *Group* and the *value* becomes *Metadata* of that *Group*.

Parameters

- **file** (path-like or file-like, optional) – The file to write the *root* to. If None then uses the value of *file* that was specified when *JSONWriter* was instantiated.
- **root** (*Root*, optional) – Write *root* in JSON format. If None then write the *Group*s and *Dataset*s in this *JSONWriter*.
- ****kwargs** – Accepts *mode*, *encoding* and *errors* keyword arguments which are passed to open(). The default *encoding* value is `'utf-8'` and the default *errors* value is `'strict'`. All additional keyword arguments are passed to json.dump. The default indentation is 2.

6.6.3 Create a New Writer

When adding a new *Writer* class to the repository the following steps should be performed. Please follow the style guide.

1. Create a fork of the repository.
2. Create a new *Writer* by following this template and save it to the msl/io/writers/ directory.

```
# import the necessary MSL-IO objects
from msl.io import Writer

class MyExampleWriter(Writer):
    """Name your class to be whatever you want, i.e., change␣
↪MyExampleWriter"""

    def write(self, file=None, root=None, **kwargs):
        """Implement your write method with the above signature.

        Parameters
        ----------
        file : path-like or file-like
            The file to write to. If None then uses the value of
            `file` that was specified when MyExampleWriter was␣
↪instantiated.
        root : Root
            Write `root` to the file. If None then write the␣
↪Groups
            and Datasets that were created using MyExampleWriter.
        **kwargs
            Optional key-value pairs.
        """
```

3. Add test cases to the tests/ directory to make sure that your Writer works as expected. It is recommended to try converting a *Root* object between your Writer and other Writers that are available to verify different file-format conversions. Also, look at the test modules that begin with `test_writer` for more examples. Run the tests using `python setup.py tests` (ideally you would run the tests for all *currently-supported versions* of Python, see also condatests.py).
4. Create a new `msl.io.writers.<name of your module from Step 2>.rst` file in docs/_api. Follow the template that is used for the other `.rst` files in this directory.
5. Add the new Writer, alphabetically, to the `.. toctree::` in docs/writers.rst. Follow the template that is used for the other Writers.
6. Add the new Writer, alphabetically, to the `.. autosummary::` in docs/api_docs.rst. Follow the template that is used for the other Writers.
7. Add yourself to `AUTHORS.rst` and add a note in `CHANGES.rst` that you created this new Writer. These files are located in the root directory of the **MSL-IO** package.
8. Build the documentation running `python setup.py docs` (view the documentation by opening the `docs/_build/html/index.html` file).
9. If running the tests pass and building the documentation show no errors/warnings then create a

pull request.

6.7 MSL-IO API Documentation

The following functions are available to read a file

read(file, **kwargs)	Read a file that has a *Reader* implemented.
read_table(file, **kwargs)	Read data in a table format from a file.
ExcelReader(file, **kwargs)	Read an Excel spreadsheet (.xls and .xlsx).
GSheetsReader(file, **kwargs)	Read a Google Sheets spreadsheet.

the following classes are available as *Writers*

HDF5Writer([file])	Create a HDF5 writer.
JSONWriter([file])	Create a JSON writer.

and general helper functions

checksum(file[, algorithm, chunk_size, ...])	Get the checksum of a file.
copy(source, destination[, overwrite, ...])	Copy a file.
git_head(directory)	Get information about the `HEAD` of a repository.
is_admin()	Check if the current process is being run as an administrator.
is_dir_accessible(path[, strict])	Check if a directory exists and is accessible.
is_file_readable(file[, strict])	Check if a file exists and is readable.
remove_write_permissions(path)	Remove all write permissions of a file.
run_as_admin([args, executable, cwd, ...])	Run a process as an administrator and return its output.
search(folder[, pattern, levels, ...])	Search for files starting from a root folder.
send_email(config, recipients[, sender, ...])	Send an email.

6.7.1 Package Structure

msl.io package

Read and write data files.

`msl.io.`**`read`**(*file*, ***kwargs*)

Read a file that has a *Reader* implemented.

Parameters

- **`file`** (path-like or file-like) – The file to read. For example, it could be a `str` representing a file system path or a stream.
- **`**kwargs`** – All keyword arguments are passed to the *Reader.can_read()* and *Reader.read()* methods.

Returns
Reader – The data from the file.

Raises
OSError – If no *Reader* exists to be able to read the specified file.

msl.io.**read_table**(*file*, ***kwargs*)

Read data in a table format from a file.

A *table* has the following properties:

1. The first row is a header.
2. All rows have the same number of columns.
3. All data values in a column have the same data type.

Parameters

- **file** (path-like or file-like) – The file to read. For example, it could be a `str` representing a file system path or a stream. If *file* is a Google Sheets spreadsheet then *file* must end with `.gsheet` even if the ID of the spreadsheet is specified.
- ****kwargs** – If the file is an Excel spreadsheet then the keyword arguments are passed to *read_table_excel()*. If a Google Sheets spreadsheet then the keyword arguments are passed to *read_table_gsheets()*. Otherwise, all keyword arguments are passed to *read_table_text()*.

Returns
Dataset – The table as a *Dataset*. The header is included as metadata.

msl.io.**version_info = version_info(major=0, minor=2, micro=0, releaselevel='dev0')**

Contains the version information as a (major, minor, micro, releaselevel) tuple.

Type
namedtuple

msl.io.base_io module

Base classes for all *Reader*s. and *Writer*s.

class msl.io.base.**Reader**(*file*)

Bases: *Root*

Parameters
file (path-like or file-like) – The file to read.

static can_read(*file*, ***kwargs*)

Whether this *Reader* can read the file specified by *file*.

Important: You must override this method.

Parameters

- **file** (path-like or file-like) – The file to check whether the *Reader* can read it.
- ****kwargs** – Key-value pairs that the *Reader* class may need when checking if it can read the *file*.

Returns
bool – Either `True` (can read) or `False` (cannot read).

static get_bytes(*file*, **args*)

Return bytes from a file.

Parameters

- **file** (path-like or file-like) – The file to read bytes from.
- ***args** (`int` or `tuple` of `int`, optional) – The position(s) in the file to retrieve bytes from.

  Examples:

  - `get_bytes(file)` → returns all bytes
  - `get_bytes(file, 5)` → returns the first 5 bytes
  - `get_bytes(file, -5)` → returns the last 5 bytes
  - `get_bytes(file, 5, 10)` → returns bytes 5 through 10 (inclusive)
  - `get_bytes(file, 3, -1)` → skips the first 2 bytes and returns the rest
  - `get_bytes(file, -8, -4)` → returns the eighth- through fourth-last bytes (inclusive)
  - `get_bytes(file, 1, -1, 2)` → returns every other byte

Returns
bytes – The bytes from the file.

static get_extension(*file*)

Return the extension of the file.

Parameters
file (path-like or file-like) – The file to get the extension of.

Returns
str – The extension, including the `'.'`.

static get_lines(*file*, **args*, ***kwargs*)

Return lines from a file.

Parameters

- **file** (path-like or file-like) – The file to read lines from.
- ***args** (`int` or `tuple` of `int`, optional) – The line(s) in the file to get.

  Examples:

  - `get_lines(file)` → returns all lines
  - `get_lines(file, 5)` → returns the first 5 lines

- `get_lines(file, -5)` → returns the last 5 lines
- `get_lines(file, 2, 4)` → returns lines 2, 3 and 4
- `get_lines(file, 4, -1)` → skips the first 3 lines and returns the rest
- `get_lines(file, 2, -2)` → skips the first and last lines and returns the rest
- `get_lines(file, -4, -2)` → returns the fourth-, third- and second-last lines
- `get_lines(file, 1, -1, 6)` → returns every sixth line in the file

****kwargs** –

- `remove_empty_lines` : `bool`

  Whether to remove all empty lines. Default is `False`.

- `encoding` : `str`

  The name of the encoding to use to decode the file. The default is `'utf-8'`.

- `errors` : `str`

  An optional string that specifies how encoding errors are to be handled. Either `'strict'` or `'ignore'`. The default is `'strict'`.

Returns

`list` of `str` – The lines from the file. Trailing whitespace is stripped from each line.

read(***kwargs*)

Read the file.

The file can be accessed by the *file* property of the *Reader*, i.e., `self.file`

Important: You must override this method.

Parameters

****kwargs** – Key-value pairs that the *Reader* class may need when reading the file.

class msl.io.base.**Root**(*file*, ***metadata*)

Bases: *Group*

The root vertex in a tree.

Parameters

- **file** (path-like or file-like) – The file object to associate with the *Root*.
- ****metadata** – Key-value pairs that can be used as *Metadata* for the *Root*.

property file

The file object that is associated with the *Root*.

Type
path-like or file-like

tree(*indent=2*)

A representation of the tree structure of all *Group*s and *Dataset*s that are in *Root*.

Parameters
indent (int, optional) – The amount of indentation to add for each recursive level.

Returns
str – The tree structure.

class msl.io.base.**Writer**(*file=None*, ***metadata*)

Bases: *Root*

Parameters

- **file** (path-like or file-like, optional) – The file to write the data to. Can also be specified in the *write()* method.
- ****metadata** – Key-value pairs that are used as *Metadata* of the *Root*.

save(*file=None*, *root=None*, ***kwargs*)

Alias for *write()*.

set_root(*root*)

Set a new *Root* for the *Writer*.

> **Attention:** This will clear the *Metadata* of the *Writer* and all *Group*s and *Dataset*s that the *Writer* currently contains. The *file* that was specified when the *Writer* was created does not change.

Parameters
root (*Root*) – The new *Root* for the *Writer*.

update_context_kwargs(***kwargs*)

When a *Writer* is used as a context manager the *write()* method is automatically called when exiting the context manager. You can specify the keyword arguments that will be passed to the *write()* method by calling *update_context_kwargs()* with the appropriate key-value pairs before the context manager exits. You can call this method multiple times since the key-value pairs get added to the underlying dict (via dict.update()) that contains all keyword arguments which are passed to the *write()* method.

write(*file=None*, *root=None*, ***kwargs*)

Write to a file.

Important: You must override this method.

Parameters

- **file** (path-like or file-like, optional) – The file to write the *root* to. If None then uses the *file* value that was specified when the *Writer* was instantiated.

- **root** (*Root*, optional) – Write the *root* object in the file format of this *Writer*. This is useful when converting between different file formats.
- ****kwargs** – Additional key-value pairs to use when writing the file.

msl.io.constants module

Constants used by MSL-IO.

msl.io.constants.**HOME_DIR = '/home/docs/.msl/io'**

The default directory where all files that are used by MSL-IO are located.

Can be overwritten by specifying a MSL_IO_HOME environment variable.

Type
str

msl.io.constants.**IS_PYTHON2 = False**

Whether Python 2 is being used.

Type
bool

msl.io.constants.**IS_PYTHON3 = True**

Whether Python 3 is being used.

Type
bool

msl.io.dataset module

A *Dataset* is essentially a numpy.ndarray with *Metadata*.

class msl.io.dataset.**Dataset**(*name*, *parent*, *read_only*, *shape=(0,)*, *dtype=<class 'float'>*, *buffer=None*, *offset=0*, *strides=None*, *order=None*, *data=None*, ***metadata*)

Bases: *Vertex*

A *Dataset* is essentially a numpy.ndarray with *Metadata*.

Do not instantiate directly. Create a new *Dataset* using *create_dataset()*.

Parameters

- **name** (str) – A name to associate with this *Dataset*.
- **parent** (*Group*) – The parent *Group* to the *Dataset*.
- **read_only** (bool) – Whether the *Dataset* is to be accessed in read-only mode.
- **shape** – See numpy.ndarray
- **dtype** – See numpy.ndarray
- **buffer** – See numpy.ndarray
- **offset** – See numpy.ndarray

- **strides** – See numpy.ndarray
- **order** – See numpy.ndarray
- **data** – If not None then it must be either a numpy.ndarray or an array-like object which will be passed to numpy.asarray(), as well as *dtype* and *order*, to be used as the underlying data.
- ****metadata** – All other key-value pairs will be used as *Metadata* for this *Dataset*.

copy(*read_only=None*)

Create a copy of this *Dataset*.

Parameters

read_only (bool, optional) – Whether the copy should be created in read-only mode. If None then creates a copy using the mode for the *Dataset* that is being copied.

Returns

Dataset – A copy of this *Dataset*.

property data

The data of the *Dataset*.

Note: You do not have to call this attribute to get access to the numpy.ndarray. You can directly call the numpy.ndarray attribute from the *Dataset* object.

For example,

```
>>> dset
<Dataset '/my_data' shape=(4, 3) dtype='<f8' (0 metadata)>
>>> dset.data
array([[ 0.,  1.,  2.],
       [ 3.,  4.,  5.],
       [ 6.,  7.,  8.],
       [ 9., 10., 11.]])
>>> dset.size
12
>>> dset.tolist()
[[0.0, 1.0, 2.0], [3.0, 4.0, 5.0], [6.0, 7.0, 8.0], [9.0, 10.0, 11.
↪0]]
>>> dset.mean(axis=0)
array([4.5, 5.5, 6.5])
>>> dset[::2]
array([[0., 1., 2.],
       [6., 7., 8.]])
```

Type

numpy.ndarray

property read_only

Whether this *Dataset* is in read-only mode.

This is equivalent to setting the WRITEABLE property in numpy.ndarray.setflags().

Type
bool

msl.io.dataset_logging module

A *Dataset* that handles logging records.

class msl.io.dataset_logging.**DatasetLogging**(*name*, *parent*, *level=0*, *attributes=None*, *logger=None*, *date_fmt=None*, ***kwargs*)

Bases: *Dataset*, Handler

A *Dataset* that handles logging records.

Do not instantiate directly. Create a new *DatasetLogging* using *create_dataset_logging()*.

Parameters

- **name** (str) – A name to associate with the *Dataset*.
- **parent** (*Group*) – The parent *Group* to the *Dataset*.
- **level** (int or str, optional) – The logging level to use.
- **attributes** (list or tuple of str, optional) – The attribute names to include in the *Dataset* for each logging record.
- **logger** (Logger, optional) – The Logger that this *DatasetLogging* object will be added to. If None then it is added to the root Logger.
- **date_fmt** (str, optional) – The datetime format code to use to represent the asctime attribute in.
- ****kwargs** – Additional keyword arguments are passed to *Dataset*. The default behaviour is to append every logging record to the *Dataset*. This guarantees that the size of the *Dataset* is equal to the number of logging records that were added to it. However, this behaviour can decrease the performance if many logging records are added often because a copy of the data in the *Dataset* is created for each logging record that is added. You can improve the performance by specifying an initial size of the *Dataset* by including a *shape* or a *size* keyword argument. This will also automatically create additional empty rows in the *Dataset*, that is proportional to the size of the *Dataset*, if the size of the *Dataset* needs to be increased. If you do this then you will want to call *remove_empty_rows()* before writing *DatasetLogging* to a file or interacting with the data in *DatasetLogging* to remove the extra rows that were created.

property attributes

The attribute names used by the *DatasetLogging* object.

Type
tuple of str

property date_fmt

The datetime format code that is used to represent the asctime attribute in.

Type
str

property logger

The Logger that this *DatasetLogging* object is added to.

Type
Logger

remove_empty_rows()

Remove empty rows from the *Dataset*.

If the *DatasetLogging* object was initialized with a *shape* or a *size* keyword argument then the size of the *Dataset* is always $\geq$ to the number of logging records that were added to it. Calling this method will remove the rows in the *Dataset* that were not from a logging record.

remove_handler()

Remove this class's Handler from the associated Logger.

After calling this method logging records are no longer added to the *Dataset*.

set_logger(*logger*)

Add this class's Handler to a Logger.

Parameters
logger (Logger) – The Logger to add this class's Handler to.

msl.io.dictionary module

An OrderedDict that can be made read only.

class msl.io.dictionary.**Dictionary**(*read_only*, ***kwargs*)

Bases: MutableMapping

A dict that can be made read only.

Parameters

- **read_only** (bool) – Whether the underlying dict should be created in read-only mode.
- ****kwargs** – Key-value pairs that are used to create the underlying dict object.

clear()

Remove all items from the dictionary.

items()

Return a new view of the dictionary's items, i.e., (key, value) pairs.

keys()

Return a new view of the dictionary's keys.

property read_only

Whether the underlying dict is in read-only mode.

Type
bool

values()

Return a new view of the dictionary's values.

msl.io.google_api module

Wrappers around Google APIs:

- *Google Drive*
- *Gmail*
- *Google Sheets*

class msl.io.google_api.**GDrive**(*account=None*, *credentials=None*, *read_only=True*, *scopes=None*)

Bases: *GoogleAPI*

Interact with Google Drive.

> **Attention:** You must follow the instructions in the prerequisites section for setting up the Drive API before you can use this class. It is also useful to be aware of the refresh token expiration policy.

Parameters

- **account** (str, optional) – Since a person may have multiple Google accounts, and multiple people may run the same code, this parameter decides which token to load to authenticate with the Google API. The value can be any text (or None) that you want to associate with a particular Google account, provided that it contains valid characters for a filename. The value that you chose when you authenticated with your *credentials* should be used for all future instances of this class to access that particular Google account. You can associate a different value with a Google account at any time (by passing in a different *account* value), but you will be asked to authenticate with your *credentials* again, or, alternatively, you can rename the token files located in *HOME_DIR* to match the new *account* value.
- **credentials** (str, optional) – The path to the *client secrets* OAuth credential file. This parameter only needs to be specified the first time that you authenticate with a particular Google account or if you delete the token file that was created when you previously authenticated.
- **read_only** (bool, optional) – Whether to interact with Google Drive in read-only mode.
- **scopes** (list of str, optional) – The list of scopes to enable for the Google API. See Drive scopes for more details. If not specified then default scopes are chosen based on the value of *read_only*.

MIME_TYPE_FOLDER = 'application/vnd.google-apps.folder'

ROOT_NAMES = ['Google Drive', 'My Drive', 'Drive']

copy(*file_id*, *folder_id=None*, *name=None*)

Copy a file.

Parameters

- **file_id** (str) – The ID of a file to copy. Folders cannot be copied.
- **folder_id** (str, optional) – The ID of the destination folder. If not specified then creates a copy in the same folder that the original file is located in. To copy the file to the *My Drive* root folder then specify `'root'` as the *folder_id*.
- **name** (str, optional) – The filename of the destination file.

Returns

str – The ID of the destination file.

create_folder(*folder*, *parent_id=None*)

Create a folder.

Makes all intermediate-level folders needed to contain the leaf directory.

Parameters

- **folder** (str) – The folder(s) to create, for example, `'folder1'` or `'folder1/folder2/folder3'`.
- **parent_id** (str, optional) – The ID of the parent folder that *folder* is relative to. If not specified then *folder* is relative to the *My Drive* root folder. If *folder* is in a *Shared drive* then you must specify the ID of a parent folder.

Returns

str – The ID of the last (right most) folder that was created.

delete(*file_or_folder_id*)

Delete a file or a folder.

Files that are in read-only mode cannot be deleted.

> **Danger:** Permanently deletes the file or folder owned by the user without moving it to the trash. If the target is a folder, then all files and sub-folders contained within the folder (that are owned by the user) are also permanently deleted.

Parameters

file_or_folder_id (str) – The ID of the file or folder to delete.

download(*file_id*, *save_to=None*, *num_retries=0*, *chunk_size=104857600*, *callback=None*)

Download a file.

Parameters

- **file_id** (str) – The ID of the file to download.
- **save_to** (path-like or file-like, optional) – The location to save the file to. If a directory is specified then the file will be saved to that directory using the filename of the remote file. To save the file with a new filename, specify

the new filename in *save_to*. Default is to save the file to the current working directory using the remote filename.

- **num_retries** (int, optional) – The number of times to retry the download. If zero (default) then attempt the request only once.
- **chunk_size** (int, optional) – The file will be downloaded in chunks of this many bytes.
- **callback** – The callback to call after each chunk of the file is downloaded. The *callback* accepts one positional argument, for example:

```
def handler(file):
    print(file.progress(), file.total_size, file.
↪resumable_progress)

drive.download('0Bwab3C2ejYSdM190b2psXy1C50P',␣
↪callback=handler)
```

empty_trash()

Permanently delete all files in the trash.

file_id(*file*, *mime_type=None*, *folder_id=None*)

Get the ID of a Google Drive file.

Parameters

- **file** (str) – The path to a Google Drive file.
- **mime_type** (str, optional) – The Drive MIME type or Media type to use to filter the results.
- **folder_id** (str, optional) – The ID of the folder that *file* is relative to. If not specified then *file* is relative to the *My Drive* root folder. If *file* is in a *Shared drive* then you must specify the ID of a parent folder.

Returns

str – The file ID.

folder_id(*folder*, *parent_id=None*)

Get the ID of a Google Drive folder.

Parameters

- **folder** (str) – The path to a Google Drive file.
- **parent_id** (str, optional) – The ID of the parent folder that *folder* is relative to. If not specified then *folder* is relative to the *My Drive* root folder. If *folder* is in a *Shared drive* then you must specify the ID of a parent folder.

Returns

str – The folder ID.

is_file(*file*, *mime_type=None*, *folder_id=None*)

Check if a file exists.

Parameters

- **file** (str) – The path to a Google Drive file.

- **mime_type** (str, optional) – The Drive MIME type or Media type to use to filter the results.
- **folder_id** (str, optional) – The ID of the folder that *file* is relative to. If not specified then *file* is relative to the *My Drive* root folder. If *file* is in a *Shared drive* then you must specify the ID of a parent folder.

Returns
bool – Whether the file exists.

is_folder(*folder*, *parent_id=None*)

Check if a folder exists.

Parameters

- **folder** (str) – The path to a Google Drive folder.
- **parent_id** (str, optional) – The ID of the parent folder that *folder* is relative to. If not specified then *folder* is relative to the *My Drive* root folder. If *folder* is in a *Shared drive* then you must specify the ID of a parent folder.

Returns
bool – Whether the folder exists.

is_read_only(*file_id*)

Returns whether the file is in read-only mode.

Parameters
file_id (str) – The ID of a file.

Returns
bool – Whether the file is in read-only mode.

move(*source_id*, *destination_id*)

Move a file or a folder.

When moving a file or folder between *My Drive* and a *Shared drive* the access permissions will change.

Moving a file or folder does not change its ID, only the ID of its *parent* changes (i.e., *source_id* will remain the same after the move).

Parameters

- **source_id** (str) – The ID of a file or folder to move.
- **destination_id** (str) – The ID of the destination folder. To move the file or folder to the *My Drive* root folder then specify `'root'` as the *destination_id*.

path(*file_or_folder_id*)

Convert an ID to a path.

Parameters
file_or_folder_id (str) – The ID of a file or folder.

Returns
str – The corresponding path of the ID.

read_only(*file_id*, *read_only*, *reason=''*)

Set a file to be in read-only mode.

Parameters

- **file_id** (str) – The ID of a file.
- **read_only** (bool) – Whether to set the file to be in read-only mode.
- **reason** (str, optional) – The reason for putting the file in read-only mode. Only used if *read_only* is True.

rename(*file_or_folder_id*, *new_name*)

Rename a file or folder.

Renaming a file or folder does not change its ID.

Parameters

- **file_or_folder_id** (str) – The ID of a file or folder.
- **new_name** (str) – The new name of the file or folder.

shared_drives()

Returns the IDs and names of all *Shared drives*.

Returns

dict – The keys are the IDs of the shared drives and the values are the names of the shared drives.

upload(*file*, *folder_id=None*, *mime_type=None*, *resumable=False*, *chunk_size=104857600*)

Upload a file.

Parameters

- **file** (str) – The file to upload.
- **folder_id** (str, optional) – The ID of the folder to upload the file to. If not specified then uploads to the *My Drive* root folder.
- **mime_type** (str, optional) – The Drive MIME type or Media type of the file (e.g., 'text/csv'). If not specified then a type will be guessed based on the file extension.
- **resumable** (bool) – Whether the upload can be resumed.
- **chunk_size** (int) – The file will be uploaded in chunks of this many bytes. Only used if *resumable* is True. Pass in a value of -1 if the file is to be uploaded in a single chunk. Note that Google App Engine has a 5MB limit on request size, so you should never set *chunk_size* to be >5MB or to -1 (if the file size is >5MB).

Returns

str – The ID of the file that was uploaded.

class msl.io.google_api.**GMail**(*account=None*, *credentials=None*, *scopes=None*)

Bases: *GoogleAPI*

Interact with Gmail.

Attention: You must follow the instructions in the prerequisites section for setting up the Gmail API before you can use this class. It is also useful to be aware of the refresh token expiration policy.

Parameters

- **account** (str, optional) – Since a person may have multiple Google accounts, and multiple people may run the same code, this parameter decides which token to load to authenticate with the Google API. The value can be any text (or None) that you want to associate with a particular Google account, provided that it contains valid characters for a filename. The value that you chose when you authenticated with your *credentials* should be used for all future instances of this class to access that particular Google account. You can associate a different value with a Google account at any time (by passing in a different *account* value), but you will be asked to authenticate with your *credentials* again, or, alternatively, you can rename the token files located in *HOME_DIR* to match the new *account* value.
- **credentials** (str, optional) – The path to the *client secrets* OAuth credential file. This parameter only needs to be specified the first time that you authenticate with a particular Google account or if you delete the token file that was created when you previously authenticated.
- **scopes** (list of str, optional) – The list of scopes to enable for the Google API. See Gmail scopes for more details. If not specified then default scopes are chosen.

profile()

Gets the authenticated user's Gmail profile.

Returns

dict – Returns the following

```
{
   'email_address': string, The authenticated user's email␣
↪address
   'messages_total': integer, The total number of messages␣
↪in the mailbox
   'threads_total': integer, The total number of threads␣
↪in the mailbox
   'history_id': string, The ID of the mailbox's current␣
↪history record
}
```

send(*recipients*, *sender='me'*, *subject=None*, *body=None*)

Send an email.

Parameters

- **recipients** (str or list of str) – The email address(es) of the recipient(s). The value 'me' can be used to indicate the authenticated user.

- **sender** (str, optional) – The email address of the sender. The value 'me' can be used to indicate the authenticated user.
- **subject** (str, optional) – The text to include in the subject field.
- **body** (str, optional) – The text to include in the body of the email. The text can be enclosed in <html></html> tags to use HTML elements to format the message.

See also:

send_email()

class msl.io.google_api.**GSheets**(*account=None*, *credentials=None*, *read_only=True*, *scopes=None*)

Bases: *GoogleAPI*

Interact with Google Sheets.

Attention: You must follow the instructions in the prerequisites section for setting up the Sheets API before you can use this class. It is also useful to be aware of the refresh token expiration policy.

Parameters

- **account** (str, optional) – Since a person may have multiple Google accounts, and multiple people may run the same code, this parameter decides which token to load to authenticate with the Google API. The value can be any text (or None) that you want to associate with a particular Google account, provided that it contains valid characters for a filename. The value that you chose when you authenticated with your *credentials* should be used for all future instances of this class to access that particular Google account. You can associate a different value with a Google account at any time (by passing in a different *account* value), but you will be asked to authenticate with your *credentials* again, or, alternatively, you can rename the token files located in *HOME_DIR* to match the new *account* value.
- **credentials** (str, optional) – The path to the *client secrets* OAuth credential file. This parameter only needs to be specified the first time that you authenticate with a particular Google account or if you delete the token file that was created when you previously authenticated.
- **read_only** (bool, optional) – Whether to interact with Google Sheets in read-only mode.
- **scopes** (list of str, optional) – The list of scopes to enable for the Google API. See Sheets scopes for more details. If not specified then default scopes are chosen based on the value of *read_only*.

MIME_TYPE = 'application/vnd.google-apps.spreadsheet'

SERIAL_NUMBER_ORIGIN = datetime.datetime(1899, 12, 30, 0, 0)

add_sheets(*names*, *spreadsheet_id*)

Add sheets to a spreadsheet.

Parameters

- **names** (str or list of str) – The name(s) of the new sheet(s) to add.
- **spreadsheet_id** (str) – The ID of the spreadsheet to add the sheet(s) to.

Returns

dict – The keys are the IDs of the new sheets and the values are the names.

append(*values*, *spreadsheet_id*, *cell=None*, *sheet=None*, *row_major=True*, *raw=False*)

Append values to a sheet.

Returns

- *values* – The value(s) to append
- **spreadsheet_id** (str) – The ID of a Google Sheets file.
- **cell** (str, optional) – The cell (top-left corner) to start appending the values to. If the cell already contains data then new rows are inserted and the values are written to the new rows. For example, `'D100'`.
- **sheet** (str, optional) – The name of a sheet in the spreadsheet to append the values to. If not specified and only one sheet exists in the spreadsheet then automatically determines the sheet name; however, it is more efficient to specify the name of the sheet.
- **row_major** (bool, optional) – Whether to append the values in row-major or column-major order.
- **raw** (bool, optional) – Determines how the values should be interpreted. If True, the values will not be parsed and will be stored as-is. If False, the values will be parsed as if the user typed them into the UI. Numbers will stay as numbers, but strings may be converted to numbers, dates, etc. following the same rules that are applied when entering text into a cell via the Google Sheets UI.

cells(*spreadsheet_id*, *ranges=None*)

Return cells from a spreadsheet.

Parameters

- **spreadsheet_id** (str) – The ID of a Google Sheets file.
- **ranges** (str or list of str, optional) – The ranges to retrieve from the spreadsheet. Examples:
  - `'Sheet1'` → return all cells from the sheet named Sheet1
  - `'Sheet1!A1:H5'` → return cells A1:H5 from the sheet named Sheet1
  - `['Sheet1!A1:H5', 'Data', 'Devices!B4:B9']` → return cells A1:H5 from the sheet named Sheet1, all cells from the sheet named Data and cells B4:B9 from the sheet named Devices

  If not specified then return all cells from all sheets.

Returns

dict – The cells from the spreadsheet. The keys are the names of the sheets and the values are a list of *GCell* objects for the specified range of each sheet.

copy(*name_or_id*, *spreadsheet_id*, *destination_spreadsheet_id*)

Copy a sheet from one spreadsheet to another spreadsheet.

Parameters

- **name_or_id** (str or int) – The name or ID of the sheet to copy.
- **spreadsheet_id** (str) – The ID of the spreadsheet that contains the sheet.
- **destination_spreadsheet_id** (str) – The ID of a spreadsheet to copy the sheet to.

Returns

int – The ID of the sheet in the destination spreadsheet.

create(*name*, *sheet_names=None*)

Create a new spreadsheet.

The spreadsheet will be created in the *My Drive* root folder. To move it to a different folder use *GDrive.create_folder()* and/or *GDrive.move()*.

Parameters

- **name** (str) – The name of the spreadsheet.
- **sheet_names** (list of str, optional) – The names of the sheets that are in the spreadsheet.

Returns

str – The ID of the spreadsheet that was created.

delete_sheets(*names_or_ids*, *spreadsheet_id*)

Delete sheets from a spreadsheet.

Parameters

- **names_or_ids** (str, int or list) – The name(s) or ID(s) of the sheet(s) to delete.
- **spreadsheet_id** (str) – The ID of the spreadsheet to delete the sheet(s) from.

rename_sheet(*name_or_id*, *new_name*, *spreadsheet_id*)

Rename a sheet.

Parameters

- **name_or_id** (str or int) – The name or ID of the sheet to rename.
- **new_name** (str) – The new name of the sheet.
- **spreadsheet_id** (str) – The ID of the spreadsheet that contains the sheet.

sheet_id(*name*, *spreadsheet_id*)

Returns the ID of a sheet.

Parameters

- **name** (str) – The name of the sheet.
- **spreadsheet_id** (str) – The ID of the spreadsheet.

Returns
int – The ID of the sheet.

sheet_names(*spreadsheet_id*)

Get the names of all sheets in a spreadsheet.

Parameters
spreadsheet_id (str) – The ID of a Google Sheets file.

Returns
tuple of str – The names of all sheets.

static to_datetime(*value*)

Convert a "serial number" date into a datetime.datetime.

Parameters
value (float) – A date in the "serial number" format.

Returns
datetime.datetime – The date converted.

values(*spreadsheet_id*, *sheet=None*, *cells=None*, *row_major=True*, *value_option=GValueOption.FORMATTED*, *datetime_option=GDateTimeOption.SERIAL_NUMBER*)

Return a range of values from a spreadsheet.

Parameters

- **spreadsheet_id** (str) – The ID of a Google Sheets file.
- **sheet** (str, optional) – The name of a sheet in the spreadsheet to read the values from. If not specified and only one sheet exists in the spreadsheet then automatically determines the sheet name; however, it is more efficient to specify the name of the sheet.
- **cells** (str, optional) – The A1 notation or R1C1 notation of the range to retrieve values from. If not specified then returns all values that are in *sheet*.
- **row_major** (bool, optional) – Whether to return the values in row-major or column-major order.
- **value_option** (str or *GValueOption*, optional) – How values should be represented in the output. If a string then it must be equal to one of the values in *GValueOption*.
- **datetime_option** (str or *GDateTimeOption*, optional) – How dates, times, and durations should be represented in the output. If a string then it must be equal to one of the values in *GDateTimeOption*. This argument is ignored if *value_option* is *GValueOption.FORMATTED*.

Returns
list – The values from the sheet.

write(*values*, *spreadsheet_id*, *cell*, *sheet=None*, *row_major=True*, *raw=False*)

Write values to a sheet.

If a cell that is being written to already contains a value, the value in that cell is overwritten with the new value.

Returns

- *values* – The value(s) to write.
- **spreadsheet_id** (`str`) – The ID of a Google Sheets file.
- **cell** (`str`, optional) – The cell (top-left corner) to start writing the values to. For example, `'C9'`.
- **sheet** (`str`, optional) – The name of a sheet in the spreadsheet to write the values to. If not specified and only one sheet exists in the spreadsheet then automatically determines the sheet name; however, it is more efficient to specify the name of the sheet.
- **row_major** (`bool`, optional) – Whether to write the values in row-major or column-major order.
- **raw** (`bool`, optional) – Determines how the values should be interpreted. If `True`, the values will not be parsed and will be stored as-is. If `False`, the values will be parsed as if the user typed them into the UI. Numbers will stay as numbers, but strings may be converted to numbers, dates, etc. following the same rules that are applied when entering text into a cell via the Google Sheets UI.

class msl.io.google_api.**GoogleAPI**(*service*, *version*, *credentials*, *scopes*, *read_only*, *account*)

Bases: `object`

Base class for all Google APIs.

close()

Close the connection to the API service.

property service

The Resource object with methods for interacting with the API service.

class msl.io.google_api.**GValueOption**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: `Enum`

Determines how values should be returned.

FORMATTED = 'FORMATTED_VALUE'

Values will be calculated and formatted in the reply according to the cell's formatting. Formatting is based on the spreadsheet's locale, not the requesting user's locale. For example, if A1 is 1.23 and A2 is =A1 and formatted as currency, then A2 would return "$1.23".

FORMULA = 'FORMULA'

Values will not be calculated. The reply will include the formulas. For example, if A1 is 1.23 and A2 is =A1 and formatted as currency, then A2 would return "=A1".

UNFORMATTED = 'UNFORMATTED_VALUE'

Values will be calculated, but not formatted in the reply. For example, if A1 is 1.23 and A2 is =A1 and formatted as currency, then A2 would return the number 1.23.

class msl.io.google_api.**GDateTimeOption**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: Enum

Determines how dates should be returned.

FORMATTED_STRING = 'FORMATTED_STRING'

Instructs date, time, datetime, and duration fields to be output as strings in their given number format (which is dependent on the spreadsheet locale).

SERIAL_NUMBER = 'SERIAL_NUMBER'

Instructs date, time, datetime, and duration fields to be output as doubles in "serial number" format, as popularized by Lotus 1-2-3. The whole number portion of the value (left of the decimal) counts the days since December 30th 1899. The fractional portion (right of the decimal) counts the time as a fraction of the day. For example, January 1st 1900 at noon would be 2.5, 2 because it's 2 days after December 30st 1899, and .5 because noon is half a day. February 1st 1900 at 3pm would be 33.625. This correctly treats the year 1900 as not a leap year.

class msl.io.google_api.**GCellType**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: Enum

The spreadsheet cell data type.

BOOLEAN = 'BOOLEAN'

CURRENCY = 'CURRENCY'

DATE = 'DATE'

DATE_TIME = 'DATE_TIME'

EMPTY = 'EMPTY'

ERROR = 'ERROR'

NUMBER = 'NUMBER'

PERCENT = 'PERCENT'

SCIENTIFIC = 'SCIENTIFIC'

STRING = 'STRING'

TEXT = 'TEXT'

TIME = 'TIME'

UNKNOWN = 'UNKNOWN'

class msl.io.google_api.**GCell**(*value*, *type*, *formatted*)

Bases: tuple

The information about a Google Sheets cell.

value

The value of the cell.

type

GCellType: The data type of *value*.

formatted

str: The formatted value (i.e., how the *value* is displayed in the cell).

msl.io.group module

A *Group* can contain sub-*Group*s and/or *Dataset*s.

class msl.io.group.**Group**(*name*, *parent*, *read_only*, ***metadata*)

Bases: *Vertex*

A *Group* can contain sub-*Group*s and/or *Dataset*s.

Do not instantiate directly. Create a new *Group* using *create_group()*.

Parameters

- **name** (str) – The name of this *Group*. Uses a naming convention analogous to UNIX file systems where each *Group* can be thought of as a directory and where every subdirectory is separated from its parent directory by the '/' character.
- **parent** (*Group*) – The parent *Group* to this *Group*.
- **read_only** (bool) – Whether the *Group* is to be accessed in read-only mode.
- ****metadata** – Key-value pairs that are used to create the *Metadata* for this *Group*.

add_dataset(*name*, *dataset*)

Add a *Dataset*.

Automatically creates the ancestor *Group*s if they do not exist.

Parameters

- **name** (str) – The name of the new *Dataset* to add.
- **dataset** (*Dataset*) – The *Dataset* to add. The *Dataset* and the *Metadata* are copied.

add_dataset_logging(*name*, *dataset_logging*)

Add a *DatasetLogging*.

Automatically creates the ancestor *Group*s if they do not exist.

Parameters

- **name** (str) – The name of the new *DatasetLogging* to add.
- **dataset_logging** (*DatasetLogging*) – The *DatasetLogging* to add. The *DatasetLogging* and the *Metadata* are copied.

add_group(*name*, *group*)

Add a *Group*.

Automatically creates the ancestor *Group*s if they do not exist.

Parameters

- **name** (str) – The name of the new *Group* to add.
- **group** (*Group*) – The *Group* to add. The *Dataset*s and *Metadata* that are contained within the *group* will be copied.

ancestors()

Get all ancestor (parent) *Group*s of this *Group*.

Yields

Group – The ancestors of this *Group*.

create_dataset(*name*, *read_only=None*, ***kwargs*)

Create a new *Dataset*.

Automatically creates the ancestor *Group*s if they do not exist.

Parameters

- **name** (str) – The name of the new *Dataset*.
- **read_only** (bool, optional) – Whether to create this *Dataset* in read-only mode. If None then uses the mode for this *Group*.
- ****kwargs** – Key-value pairs that are passed to *Dataset*.

Returns

Dataset – The new *Dataset* that was created.

create_dataset_logging(*name*, *level='INFO'*, *attributes=None*, *logger=None*, *date_fmt=None*, ***kwargs*)

Create a *Dataset* that handles logging records.

Automatically creates the ancestor *Group*s if they do not exist.

Parameters

- **name** (str) – A name to associate with the *Dataset*.
- **level** (int or str, optional) – The logging level to use.
- **attributes** (list or tuple of str, optional) – The attribute names to include in the *Dataset* for each logging record. If None then uses asctime, levelname, name, and message.
- **logger** (Logger, optional) – The Logger that the *DatasetLogging* object will be added to. If None then it is added to the root Logger.
- **date_fmt** (str, optional) – The datetime format code to use to represent the asctime attribute in. If None then uses the ISO 8601 format '%Y-%m-%dT%H:%M:%S.%f'.
- ****kwargs** – Additional keyword arguments are passed to *Dataset*. The default behaviour is to append every logging record to the *Dataset*. This guarantees that the size of the *Dataset* is equal to the number of logging records that were added to it. However, this behaviour can decrease the performance if many logging records are added often because a copy of the data in the *Dataset* is created for each logging record that is added. You can improve the performance by specifying an initial size of the *Dataset* by including a *shape* or a *size* keyword argument. This will also automatically

create additional empty rows in the *Dataset*, that is proportional to the size of the *Dataset*, if the size of the *Dataset* needs to be increased. If you do this then you will want to call *remove_empty_rows()* before writing *DatasetLogging* to a file or interacting with the data in *DatasetLogging* to remove the extra rows that were created.

Returns

DatasetLogging – The *DatasetLogging* that was created.

Examples

```
>>> import logging
>>> from msl.io import JSONWriter
>>> logger = logging.getLogger('my_logger')
>>> root = JSONWriter()
>>> log_dset = root.create_dataset_logging('log')
>>> logger.info('hi')
>>> logger.error('cannot do that!')
>>> log_dset.data
array([(..., 'INFO', 'my_logger', 'hi'), (..., 'ERROR', 'my_logger',
↪'cannot do that!')],
      dtype=[('asctime', 'O'), ('levelname', 'O'), ('name', 'O'), (
↪'message', 'O')])
```

Get all ERROR logging records

```
>>> errors = log_dset[log_dset['levelname'] == 'ERROR']
>>> print(errors)
[(..., 'ERROR', 'my_logger', 'cannot do that!')]
```

Stop the *DatasetLogging* object from receiving logging records

```
>>> log_dset.remove_handler()
```

create_group(*name*, *read_only=None*, ***metadata*)

Create a new *Group*.

Automatically creates the ancestor *Group*s if they do not exist.

Parameters

- **name** (str) – The name of the new *Group*.
- **read_only** (bool, optional) – Whether to create this *Group* in read-only mode. If None then uses the mode for this *Group*.
- ****metadata** – Key-value pairs that are used to create the *Metadata* for this *Group*.

Returns

Group – The new *Group* that was created.

datasets(*exclude=None*, *include=None*, *flags=0*)

Get the *Dataset*s in this *Group*.

Parameters

- **exclude** (str, optional) – A regex pattern to use to exclude *Dataset*s. The re.search() function is used to compare the *exclude* regex pattern with the *name* of each *Dataset*. If there is a match, the *Dataset* is not yielded.
- **include** (str, optional) – A regex pattern to use to include *Dataset*s. The re.search() function is used to compare the *include* regex pattern with the *name* of each *Dataset*. If there is a match, the *Dataset* is yielded.
- **flags** (int, optional) – Regex flags that are passed to re.compile().

Yields

Dataset – The filtered *Dataset*s based on the *exclude* and *include* regex patterns. The *exclude* pattern has more precedence than the *include* pattern if there is a conflict.

descendants()

Get all descendant (children) *Group*s of this *Group*.

Yields

Group – The descendants of this *Group*.

groups(*exclude=None*, *include=None*, *flags=0*)

Get the sub-*Group*s of this *Group*.

Parameters

- **exclude** (str, optional) – A regex pattern to use to exclude *Group*s. The re.search() function is used to compare the *exclude* regex pattern with the *name* of each *Group*. If there is a match, the *Group* is not yielded.
- **include** (str, optional) – A regex pattern to use to include *Group*s. The re.search() function is used to compare the *include* regex pattern with the *name* of each *Group*. If there is a match, the *Group* is yielded.
- **flags** (int, optional) – Regex flags that are passed to re.compile().

Yields

Group – The filtered *Group*s based on the *exclude* and *include* regex patterns. The *exclude* pattern has more precedence than the *include* pattern if there is a conflict.

static is_dataset(*obj*)

Test whether an object is a *Dataset*.

Parameters

obj (object) – The object to test.

Returns

bool – Whether *obj* is an instance of *Dataset*.

static is_dataset_logging(*obj*)

Test whether an object is a *DatasetLogging*.

Parameters

obj (object) – The object to test.

Returns

bool – Whether *obj* is an instance of *DatasetLogging*.

static is_group(*obj*)

Test whether an object is a *Group*.

Parameters
obj (object) – The object to test.

Returns
bool – Whether *obj* is an instance of *Group*.

remove(*name*)

Remove a *Group* or a *Dataset*.

Parameters
name (str) – The name of the *Group* or *Dataset* to remove.

Returns
Group, *Dataset* or None – The *Group* or *Dataset* that was removed or None if there was no *Group* or *Dataset* with the specified *name*.

require_dataset(*name*, *read_only=None*, ***kwargs*)

Require that a *Dataset* exists.

If the *Dataset* exists then it will be returned if it does not exist then it is created.

Automatically creates the ancestor *Group*s if they do not exist.

Parameters

- **name** (str) – The name of the *Dataset*.
- **read_only** (bool, optional) – Whether to create this *Dataset* in read-only mode. If None then uses the mode for this *Group*.
- ****kwargs** – Key-value pairs that are passed to *Dataset*.

Returns
Dataset – The *Dataset* that was created or that already existed.

require_dataset_logging(*name*, *level='INFO'*, *attributes=None*, *logger=None*, *date_fmt=None*, ***kwargs*)

Require that a *Dataset* exists for handling logging records.

If the *DatasetLogging* exists then it will be returned if it does not exist then it is created.

Automatically creates the ancestor *Group*s if they do not exist.

Parameters

- **name** (str) – A name to associate with the *Dataset*.
- **level** (int or str, optional) – The logging level to use.
- **attributes** (list or tuple of str, optional) – The attribute names to include in the *Dataset* for each logging record. If the *Dataset* exists and if *attributes* are specified, and they do not match those of the existing *Dataset*, then a ValueError is raised. If None and the *Dataset* does not exist then uses asctime, levelname, name, and message.
- **logger** (Logger, optional) – The Logger that the *DatasetLogging* object will be added to. If None then it is added to the root Logger.

- **date_fmt** (str, optional) – The datetime format code to use to represent the asctime attribute in. If None then uses the ISO 8601 format '%Y-%m-%dT%H:%M:%S.%f'.
- ****kwargs** – Additional keyword arguments are passed to *Dataset*. The default behaviour is to append every logging record to the *Dataset*. This guarantees that the size of the *Dataset* is equal to the number of logging records that were added to it. However, this behaviour can decrease the performance if many logging records are added often because a copy of the data in the *Dataset* is created for each logging record that is added. You can improve the performance by specifying an initial size of the *Dataset* by including a *shape* or a *size* keyword argument. This will also automatically create additional empty rows in the *Dataset*, that is proportional to the size of the *Dataset*, if the size of the *Dataset* needs to be increased. If you do this then you will want to call *remove_empty_rows()* before writing *DatasetLogging* to a file or interacting with the data in *DatasetLogging* to remove the extra rows that were created.

Returns

DatasetLogging – The *DatasetLogging* that was created or that already existed.

require_group(*name*, *read_only=None*, ***metadata*)

Require that a *Group* exists.

If the *Group* exists then it will be returned if it does not exist then it is created.

Automatically creates the ancestor *Group*s if they do not exist.

Parameters

- **name** (str) – The name of the *Group*.
- **read_only** (bool, optional) – Whether to return the *Group* in read-only mode. If None then uses the mode for this *Group*.
- ****metadata** – Key-value pairs that are used as *Metadata* for this *Group*.

Returns

Group – The *Group* that was created or that already existed.

msl.io.metadata module

Provides information about other data.

class msl.io.metadata.**Metadata**(*read_only*, *vertex_name*, ***kwargs*)

Bases: *Dictionary*

Provides information about other data.

Do not instantiate directly. A *Metadata* object is created automatically when *create_dataset()* or *create_group()* is called.

Parameters

- **read_only** (bool) – Whether *Metadata* is to be accessed in read-only mode.

- **vertex_name** (str) – The name of the *Vertex* that *Metadata* is associated with.
- ****kwargs** – Key-value pairs that will be used to create the *Dictionary*.

copy(*read_only=None*)

Create a copy of the *Metadata*.

Parameters

read_only (bool, optional) – Whether the copy should be created in read-only mode. If None then creates a copy using the mode for the *Metadata* that is being copied.

Returns

Metadata – A copy of the *Metadata*.

fromkeys(*seq*, *value=None*, *read_only=None*)

Create a new *Metadata* object with keys from *seq* and values set to *value*.

Parameters

- **seq** – Any iterable object that contains the names of the keys.
- **value** (object, optional) – The default value to use for each key.
- **read_only** (bool, optional) – Whether the returned object should be created in read-only mode. If None then uses the mode for the *Metadata* that is used to call this method.

Returns

Metadata – A new *Metadata* object.

msl.io.tables module

Read a data table from a file.

msl.io.tables.**extension_delimiter_map = {'.csv': ','}**

The delimiter to use to separate columns in a table based on the file extension.

If the *delimiter* is not specified when calling the *read_table()* function then this extension-delimiter map is used to determine the value of the *delimiter*. If the file extension is not in the map then the value of the *delimiter* is None (i.e., split columns by any whitespace).

Examples

You can customize your own map by adding key-value pairs

```
>>> from msl.io import extension_delimiter_map
>>> extension_delimiter_map['.txt'] = '\t'
```

Type

dict

msl.io.tables.**read_table_excel**(*file*, *cells=None*, *sheet=None*, *as_datetime=True*, *dtype=None*, ***kwargs*)

Read a data table from an Excel spreadsheet.

A *table* has the following properties:

1. The first row is a header.
2. All rows have the same number of columns.
3. All data values in a column have the same data type.

Parameters

- **file** (path-like or file-like) – The file to read.
- **cells** (`str`, optional) – The cells to read. For example, `C9` will start at cell C9 and include all values until the end of the spreadsheet, `A:C` includes all rows in columns A, B and C, and, `C9:G20` includes values from only the specified cells. If not specified then returns all values from the specified *sheet*.
- **sheet** (`str`, optional) – The name of the sheet to read the data from. If there is only one sheet in the workbook then you do not need to specify the name of the sheet.
- **as_datetime** (`bool`, optional) – Whether dates should be returned as `datetime` or `date` objects. If `False` then dates are returned as a `str`.
- **dtype** (`object`, optional) – If specified then it must be able to be converted to a `dtype` object.
- ****kwargs** – All additional keyword arguments are passed to `open_workbook()`. Can use an *encoding* keyword argument as an alias for *encoding_override*.

Returns

Dataset – The table as a *Dataset*. The header is included in the *Metadata*.

msl.io.tables.**read_table_gsheets**(*file*, *cells=None*, *sheet=None*, *as_datetime=True*, *dtype=None*, ***kwargs*)

Read a data table from a Google Sheets spreadsheet.

> **Attention:** You must have already performed the instructions specified in *GDrive* and in *GSheets* to be able to use this function.

A *table* has the following properties:

1. The first row is a header.
2. All rows have the same number of columns.
3. All data values in a column have the same data type.

Parameters

- **file** (path-like or file-like) – The file to read. Can be the ID of a Google Sheets spreadsheet.

- **cells** (str, optional) – The cells to read. For example, `C9` will start at cell C9 and include all values until the end of the spreadsheet, `A:C` includes all rows in columns A, B and C, and, `C9:G20` includes values from only the specified cells. If not specified then returns all values from the specified *sheet*.
- **sheet** (str, optional) – The name of the sheet to read the data from. If there is only one sheet in the spreadsheet then you do not need to specify the name of the sheet.
- **as_datetime** (bool, optional) – Whether dates should be returned as `datetime` or `date` objects. If `False` then dates are returned as a `str`.
- **dtype** (`object`, optional) – If specified then it must be able to be converted to a `dtype` object.
- ****kwargs** – All additional keyword arguments are passed to *GSheetsReader*.

Returns

Dataset – The table as a *Dataset*. The header is included in the *Metadata*.

`msl.io.tables.`**read_table_text**(*file*, ***kwargs*)

Read a data table from a text-based file.

A *table* has the following properties:

1. The first row is a header.
2. All rows have the same number of columns.
3. All data values in a column have the same data type.

Parameters

- **file** (path-like or file-like) – The file to read.
- ****kwargs** – All keyword arguments are passed to `loadtxt()`. If the *delimiter* is not specified and the *file* has `csv` as the file extension then the *delimiter* is automatically set to be `','`.

Returns

Dataset – The table as a *Dataset*. The header is included in the *Metadata*.

msl.io.utils module

General functions.

`msl.io.utils.`**checksum**(*file*, *algorithm='sha256'*, *chunk_size=65536*, *shake_length=256*)

Get the checksum of a file.

A checksum is a sequence of numbers and letters that act as a fingerprint for a file against which later comparisons can be made to detect errors or changes in the file. It can be used to verify the integrity of the data.

Parameters

- **file** (path-like or file object) – A file to get the checksum of.
- **algorithm** (str, optional) – The hash algorithm to use to compute the checksum. See `hashlib` for more details.

- **chunk_size** (int, optional) – The number of bytes to read at a time from the file. It is useful to tweak this parameter when reading a large file to improve performance.
- **shake_length** (int, optional) – The digest length to use for the SHAKE algorithm. See hashlib.shake.hexdigest() for more details.

Returns
str – The checksum containing only hexadecimal digits.

msl.io.utils.**copy**(*source*, *destination*, *overwrite=False*, *include_metadata=True*)

Copy a file.

Parameters

- **source** (path-like object) – The path to a file to copy.
- **destination** (path-like object) – A directory to copy the file to or a full path (i.e., includes the basename). If the directory does not exist then it, and all intermediate directories, will be created.
- **overwrite** (bool, optional) – Whether to overwrite the *destination* file if it already exists. If *destination* already exists and *overwrite* is False then a FileExistsError is raised.
- **include_metadata** (bool, optional) – Whether to also copy information such as the file permissions, the latest access time and latest modification time with the file.

Returns
str – The path to where the file was copied.

msl.io.utils.**get_basename**(*obj*)

Get the basename() of a file.

Parameters
obj (path-like or file-like) – The object to get the basename() of. If the object does not support the basename() function then the __name__ of the *obj* is returned.

Returns
str – The basename of *obj*.

msl.io.utils.**git_head**(*directory*)

Get information about the HEAD of a repository.

This function requires that git is installed and that it is available on PATH.

Parameters
directory (str) – A directory that is under version control.

Returns
dict or None – Information about the most recent commit on the current branch. If *directory* is not a directory that is under version control then returns None.

msl.io.utils.**is_admin**()

Check if the current process is being run as an administrator.

Returns
bool – Whether the current process is being run as an administrator.

msl.io.utils.**is_dir_accessible**(*path*, *strict=False*)

Check if a directory exists and is accessible.

An accessible directory is one that the user has permission to access.

Parameters

- **path** (str) – The directory to check.
- **strict** (bool, optional) – Whether to raise the exception (if one occurs).

Returns

bool – Whether the directory exists and is accessible.

msl.io.utils.**is_file_readable**(*file*, *strict=False*)

Check if a file exists and is readable.

Parameters

- **file** (str) – The file to check.
- **strict** (bool, optional) – Whether to raise the exception (if one occurs).

Returns

bool – Whether the file exists and is readable.

msl.io.utils.**register**(*reader_class*)

Use as a decorator to register a *Reader* subclass.

See *Create a New Reader* for an example on how to use @register decorator.

Parameters

reader_class (*Reader*) – A *Reader* subclass.

Returns

Reader – The *Reader*.

msl.io.utils.**remove_write_permissions**(*path*)

Remove all write permissions of a file.

On Windows, this function will set the file attribute to be read only.

On linux and macOS, write permission is removed for the User, Group and Others. The read and execute permissions are preserved.

Parameters

path (path-like object) – The path to remove the write permissions of.

msl.io.utils.**run_as_admin**(*args=None*, *executable=None*, *cwd=None*, *capture_stderr=False*, *blocking=True*, *show=False*, ***kwargs*)

Run a process as an administrator and return its output.

Parameters

- **args** (str or list of str, optional) – A sequence of program arguments or else a single string. Providing a sequence of arguments is generally preferred, as it allows the module to take care of any required escaping and quoting of arguments (e.g., to permit spaces in file names).
- **executable** (str, optional) – The executable to pass the *args* to.
- **cwd** (str, optional) – The working directory for the elevated process.

- **capture_stderr** (bool, optional) – Whether to send the stderr stream to stdout.
- **blocking** (bool, optional) – Whether to wait for the process to finish before returning to the calling program.
- **show** (bool, optional) – Whether to show the elevated console (Windows only). If True then the stdout stream of the process is not captured.
- **kwargs** – If the current process already has admin privileges or if the operating system is not Windows then all additional keyword arguments are passed to check_output(). Otherwise, only a *timeout* keyword argument is used (Windows).

Returns

bytes, int or Popen – The returned object depends on whether the process is executed in blocking or non-blocking mode. If blocking then bytes are returned (the stdout stream of the process). If non-blocking, then the returned object will either be the Popen instance that is running the process (POSIX) or an int which is the process ID (Windows).

Examples

Import the modules

```
>>> import sys
>>> from msl.io import run_as_admin
```

Run a shell script

```
>>> run_as_admin(['./script.sh', '--message', 'hello world'])
```

Run a Python script

```
>>> run_as_admin([sys.executable, 'script.py', '--verbose'], cwd='D:\\My␣
↪Scripts')
```

Create a service in the Windows registry and in the Service Control Manager database

```
>>> run_as_admin(['sc', 'create', 'MyLogger', 'binPath=', 'C:\\logger.exe
↪', 'start=', 'auto'])
```

msl.io.utils.**search**(*folder*, *pattern=None*, *levels=0*, *regex_flags=0*, *exclude_folders=None*, *ignore_permission_error=True*, *ignore_hidden_folders=True*, *follow_symlinks=False*)

Search for files starting from a root folder.

Parameters

- **folder** (str) – The root folder to begin searching for files.
- **pattern** (str, optional) – A regex string to use to filter the filenames. If None then no filtering is applied and all files are yielded. Examples:
  - r'data' → find all files with the word data in the filename

  - `r'\.png$'` → find all files with the extension `.png`
  - `r'\.jpe*g$'` → find all files with the extension `.jpeg` or `.jpg`
- **levels** (int, optional) – The number of sub-folder levels to recursively search for files. If None then search all sub-folders.
- **regex_flags** (int, optional) – The flags to use to compile regex strings.
- **exclude_folders** (str or list of str, optional) – The pattern of folder names to exclude from the search. Can be a regex string. If None then include all folders in the search. Examples:
  - `r'bin'` → exclude all folders that contain the word `bin`
  - `r'^My'` → exclude all folders that start with the letters `My`
  - `[r'bin', r'^My']` which is equivalent to `r'(bin|^My')` → exclude all folders that contain the word `bin` or start with the letters `My`
- **ignore_permission_error** (bool, optional) – Whether to ignore PermissionError exceptions when reading the items within a folder.
- **ignore_hidden_folders** (bool, optional) – Whether to ignore hidden folders from the search. A hidden folder starts with a `.` (a dot).
- **follow_symlinks** (bool, optional) – Whether to search for files by following symbolic links.

Yields

str – The path to a file.

msl.io.utils.**send_email**(*config*, *recipients*, *sender=None*, *subject=None*, *body=None*)

Send an email.

Parameters

- **config** – A path-like object or file-like object of an INI-style configuration file that contains information on how to send an email. There are two ways to send an email – Gmail API or SMTP server.

  An example INI file to use the Gmail API is the following (see *GMail* for more details). Although all key-value pairs are optional, a `[gmail]` section must exist to use the Gmail API.

```
[gmail]
account = work [default: None]
credentials = path/to/client_secrets.json [default: None]
scopes =       [default: None]
  https://www.googleapis.com/auth/gmail.send
  https://www.googleapis.com/auth/gmail.metadata
domain = @gmail.com [default: None]
```

  An example INI file for an SMTP server is the following. Only the *host* and *port* key-value pairs are required.

```
[smtp]
host = hostname or IP address of the SMTP server
```

(continues on next page)

(continued from previous page)

```
port = port number to connect to on the SMTP server
starttls = true|yes|1|on -or- false|no|0|off [default:␣
↪false]
username = the username to authenticate with [default:␣
↪None]
password = the password for username [default: None]
domain = @company.com [default: None]
```

> **Warning:** Since this information is specified in plain text in the configuration file, you should set the file permissions provided by your operating system to ensure that your authentication credentials are safe.

- **recipients** (str or list of str) – The email address(es) of the recipient(s). Can omit the `@domain.com` part if a `domain` key is specified in the *config* file. Can be the value `'me'` if sending an email to yourself via Gmail.
- **sender** (str, optional) – The email address of the sender. Can omit the `@domain.com` part if a `domain` key is specified in the *config* file. If not specified then it equals the value of the first *recipient* if using SMTP or the value `'me'` if using Gmail.
- **subject** (str, optional) – The text to include in the subject field.
- **body** (str, optional) – The text to include in the body of the email. The text can be enclosed in `<html></html>` tags to use HTML elements to format the message.

msl.io.vertex module

A vertex in a tree.

class msl.io.vertex.**Vertex**(*name*, *parent*, *read_only*, ***metadata*)

Bases: *Dictionary*

A vertex in a tree.

Parameters

- **name** (str) – The name of this vertex.
- **parent** (*Group*) – The parent of this vertex.
- **read_only** (bool) – Whether this vertex is in read-only mode.
- ****metadata** – Key-value pairs that are used to create the *Metadata* for this *Vertex*.

add_metadata(***metadata*)

Add key-value pairs to the *Metadata* for this *Vertex*.

property metadata

The metadata associated with this *Vertex*.

Type
Metadata

property name

The name of this *Vertex*.

Type
str

property parent

The parent of this *Vertex*.

Type
Group

property read_only

Whether this *Vertex* is in read-only mode.

Setting this value will also update all sub-*Group*s and sub-*Dataset*s to be in the same mode.

Type
bool

msl.io.readers package

Submodules

msl.io.readers.excel module

Read an Excel spreadsheet (.xls and .xlsx).

class msl.io.readers.excel.**ExcelReader**(*file*, ***kwargs*)

Bases: *Spreadsheet*

Read an Excel spreadsheet (.xls and .xlsx).

This class simply provides a convenience for reading information from Excel spreadsheets. It is not registered as a *Reader* because the information in an Excel spreadsheet is unstructured and therefore one cannot generalize how to parse an Excel spreadsheet to create a *Root*.

Parameters

- **file** (str) – The location of an Excel spreadsheet on a local hard drive or on a network.
- ****kwargs** – All keyword arguments are passed to open_workbook(). Can use an *encoding* keyword argument as an alias for *encoding_override*. The default *on_demand* value is True.

Examples

```
>>> from msl.io import ExcelReader
>>> excel = ExcelReader('lab_environment.xlsx')
```

close()

Calls `release_resources()`.

read(*cell=None*, *sheet=None*, *as_datetime=True*)

Read values from the Excel spreadsheet.

Parameters

- **cell** (`str`, optional) – The cell(s) to read. For example, `C9` will return a single value and `C9:G20` will return all values in the specified range. If not specified then returns all values in the specified *sheet*.
- **sheet** (`str`, optional) – The name of the sheet to read the value(s) from. If there is only one sheet in the spreadsheet then you do not need to specify the name of the sheet.
- **as_datetime** (`bool`, optional) – Whether dates should be returned as `datetime` or `date` objects. If `False` then dates are returned as an ISO 8601 string.

Returns

The value(s) of the requested cell(s).

Examples

```
>>> excel.read()
[('temperature', 'humidity'), (20.33, 49.82), (20.23, 46.06), (20.41,
↪ 47.06), (20.29, 48.32)]
>>> excel.read('B2')
49.82
>>> excel.read('A:A')
[('temperature',), (20.33,), (20.23,), (20.41,), (20.29,)]
>>> excel.read('A1:B1')
[('temperature', 'humidity')]
>>> excel.read('A2:B4')
[(20.33, 49.82), (20.23, 46.06), (20.41, 47.06)]
```

sheet_names()

Get the names of all sheets in the Excel spreadsheet.

Returns

`tuple` of `str` – The names of all sheets.

property workbook

The workbook instance.

Type

`Book`

msl.io.readers.gsheets module

Read a Google Sheets spreadsheet.

class msl.io.readers.gsheets.**GSheetsReader**(*file*, ***kwargs*)

Bases: *Spreadsheet*

Read a Google Sheets spreadsheet.

This class simply provides a convenience for reading information from Google spreadsheets. It is not registered as a *Reader* because the information in a spreadsheet is unstructured and therefore one cannot generalize how to parse a spreadsheet to create a *Root*.

Parameters

- **file** (str) – The ID or path of a Google Sheets spreadsheet.
- ****kwargs** – All keyword arguments are passed to *GSheets*.

Examples

```
>>> from msl.io import GSheetsReader
>>> sheets = GSheetsReader('Google Drive/registers/equipment.gsheet')
>>> sheets = GSheetsReader('1TI3pM-534SZ5DQTEZ-7HCI04648f8ZpLGbfHWJu9FSo')
```

close()

Close the connection to the GSheet API service.

New in version 0.2.

read(*cell=None*, *sheet=None*, *as_datetime=True*)

Read values from the Google Sheets spreadsheet.

Parameters

- **cell** (str, optional) – The cell(s) to read. For example, `C9` will return a single value and `C9:G20` will return all values in the specified range. If not specified then returns all values in the specified *sheet*.
- **sheet** (str, optional) – The name of the sheet to read the value(s) from. If there is only one sheet in the spreadsheet then you do not need to specify the name of the sheet.
- **as_datetime** (bool, optional) – Whether dates should be returned as `datetime` or `date` objects. If `False` then dates are returned as a string in the format of the spreadsheet cell.

Returns

The value(s) of the requested cell(s).

Examples

```
>>> sheets.read()
[('temperature', 'humidity'), (20.33, 49.82), (20.23, 46.06), (20.41,
↪ 47.06), (20.29, 48.32)]
>>> sheets.read('B2')
49.82
>>> sheets.read('A:A')
[('temperature',), (20.33,), (20.23,), (20.41,), (20.29,)]
>>> sheets.read('A1:B1')
[('temperature', 'humidity')]
>>> sheets.read('A2:B4')
[(20.33, 49.82), (20.23, 46.06), (20.41, 47.06)]
```

sheet_names()

Get the names of all sheets in the Google Sheets spreadsheet.

Returns

tuple of str – The names of all sheets.

msl.io.readers.spreadsheet module

Generic class for spreadsheets.

class msl.io.readers.spreadsheet.**Spreadsheet**(*file*)

Bases: object

Generic class for spreadsheets.

Parameters

file (str) – The location of the spreadsheet on a local hard drive or on a network.

property file

The location of the spreadsheet on a local hard drive or on a network.

Type

str

read(*cell=None*, *sheet=None*, *as_datetime=True*)

Read values from the spreadsheet.

Parameters

- **cell** (str, optional) – The cell(s) to read. For example, C9 will return a single value and C9:G20 will return all values in the specified range. If not specified then returns all values in the specified *sheet*.
- **sheet** (str, optional) – The name of the sheet to read the value(s) from. If there is only one sheet in the spreadsheet then you do not need to specify the name of the sheet.
- **as_datetime** (bool, optional) – Whether dates should be returned as datetime or date objects. If False then dates are returned as a string.

Returns
The value(s) of the requested cell(s).

sheet_names()

Get the names of all sheets in the spreadsheet.

Returns
tuple of str – The names of all sheets.

static to_indices(*cell*)

Convert a string representation of a cell to row and column indices.

Parameters
cell (str) – The cell. Can be letters only (a column) or letters and a number (a column and a row).

Returns
tuple – The (row_index, column_index). If *cell* does not contain a row number then the row index is None. The row and column index are zero based.

Examples

```
>>> to_indices('A')
(None, 0)
>>> to_indices('A1')
(0, 0)
>>> to_indices('AA10')
(9, 26)
>>> to_indices('AAA111')
(110, 702)
>>> to_indices('MSL123456')
(123455, 9293)
>>> to_indices('BIPM')
(None, 41664)
```

static to_letters(*index*)

Convert a column index to column letters.

Parameters
index (int) – The column index (zero based).

Returns
str – The corresponding spreadsheet column letter(s).

Examples

```
>>> to_letters(0)
'A'
>>> to_letters(1)
'B'
>>> to_letters(26)
'AA'
>>> to_letters(702)
'AAA'
>>> to_letters(494264)
'ABCDE'
```

static to_slices(*cells*, *row_step=None*, *column_step=None*)

Convert a range of cells to slices of row and column indices.

Parameters

- **cells** (str) – The cells. Can be letters only (a column) or letters and a number (a column and a row).
- **row_step** (int, optional) – The step-by value for the row slice.
- **column_step** (int, optional) – The step-by value for the column slice.

Returns

- slice – The row slice.
- slice – The column slice.

Examples

```
>>> to_slices('A:A')
(slice(0, None, None), slice(0, 1, None))
>>> to_slices('A:H')
(slice(0, None, None), slice(0, 8, None))
>>> to_slices('B2:M10')
(slice(1, 10, None), slice(1, 13, None))
>>> to_slices('A5:M100', row_step=2, column_step=4)
(slice(4, 100, 2), slice(0, 13, 4))
```

msl.io.writers package

Submodules

6.8 Accessing Keys as Class Attributes

In order to access a dictionary *key* as a class attribute, for a *Group* or a *Metadata* object, or the *fieldnames* of a numpy array in a *Dataset*, then the following naming rules must be followed:

- the name matches the regex pattern `^[a-zA-Z][a-zA-Z0-9_]*$` – which states that the name must begin with a letter and is followed by zero or more alphanumeric characters or underscores
- the name cannot be equal to any of the following:
  - clear
  - copy
  - fromkeys
  - get
  - read_only
  - items
  - keys
  - pop
  - popitem
  - setdefault
  - update
  - values

6.9 License

6.10 Developers

- Joseph Borbely <joseph.borbely@measurement.govt.nz>
- Rebecca Hawke <rebecca.hawke@measurement.govt.nz>

6.11 Release Notes

6.11.1 Version 0.2.0 (in development)

- Added
  - `GSheetsReader` and `ExcelReader` can now be used as a context manager
  - `GSheetsReader.close` method
  - a *TEXT* member to the `GCellType` enum
- Removed
  - Support for Python 2.7, 3.5, 3.6 and 3.7

6.11.2 Version 0.1.0 (2023-06-16)

Initial release.

It is also the last release to support Python 2.7, 3.5, 3.6 and 3.7

CHAPTER

SEVEN

INDEX

- modindex

PYTHON MODULE INDEX

m

INDEX

N

P

R

S

T

U

V

W