Mt. Data

Release 2.0.1

George Lesica, Smai Fullerton, Eric Dietrich

Sep 25, 2023

CONTENTS:

Mt. Data (pronounced “mount data”, like “Mount Everest”) is a tool to help people collect and store (mountains of) public data from a variety of sources.

CHAPTER

ONE

DEVELOPER DOCUMENTATION

It is relatively straightforward to extend Mt. Data in various ways. The documentation below walks through several common tasks such as adding a new dataset.

1.1 mtdata

1.1.1 mtdata package

Subpackages

mtdata.datasets package

Submodules

mtdata.datasets.air_quality module

class mtdata.datasets.air_quality.**AirQuality**

Bases: *Dataset*

Air quality data for Montana.

property dedup_facets: Iterable[str]

Fields used to determine which rows should be compared for de-duplication.

For example, if the data are sensor readings for various locations, the field that indicates the location would be listed here. That way, we don't drop a new reading from a different location just because it occurred at the same time as a reading from a different location.

Uses the transformed version of the field names.

property dedup_fields: Iterable[str]

Fields used to compare rows for de-duplication. This is likely to be some kind of timestamp, but that depends on the kind of data.

For example, if the data are sensor readings and each row is timestamped based on when the reading occurred, then that field will be listed here because two or more fetches might retrieve the same reading instance.

Uses the transformed version of the field names.

fetch() → *FetchResult*

Fetch new data from the source (generally the web).

static name() → str

The dataset name, which is used in the UI and for things like file and table names.

property transformer: *Transformer*

The transformer to be applied to each row that is fetched from the data source before it is stored.

mtdata.datasets.missoula_911 module

class mtdata.datasets.missoula_911.**Missoula911**

Bases: *Dataset*

Record of 911 events for Missoula city and county in Montana.

property dedup_facets: Iterable[str]

Fields used to determine which rows should be compared for de-duplication.

For example, if the data are sensor readings for various locations, the field that indicates the location would be listed here. That way, we don't drop a new reading from a different location just because it occurred at the same time as a reading from a different location.

Uses the transformed version of the field names.

property dedup_fields: Iterable[str]

Fields used to compare rows for de-duplication. This is likely to be some kind of timestamp, but that depends on the kind of data.

For example, if the data are sensor readings and each row is timestamped based on when the reading occurred, then that field will be listed here because two or more fetches might retrieve the same reading instance.

Uses the transformed version of the field names.

fetch() → *FetchResult*

Fetch new data from the source (generally the web).

static name() → str

The dataset name, which is used in the UI and for things like file and table names.

property transformer: *Transformer*

The transformer to be applied to each row that is fetched from the data source before it is stored.

mtdata.datasets.mt_covid_counts module

class mtdata.datasets.mt_covid_counts.**CovidCounts**

Bases: *Dataset*

Covid-19 case data for Montana.

Source front-end display: https://www.arcgis.com/apps/MapSeries/index.html?appid=7c34f3412536439491adcc2103421d4b

property dedup_facets: Iterable[str]

Fields used to determine which rows should be compared for de-duplication.

For example, if the data are sensor readings for various locations, the field that indicates the location would be listed here. That way, we don't drop a new reading from a different location just because it occurred at the same time as a reading from a different location.

Uses the transformed version of the field names.

property dedup_fields: Iterable[str]

Fields used to compare rows for de-duplication. This is likely to be some kind of timestamp, but that depends on the kind of data.

For example, if the data are sensor readings and each row is timestamped based on when the reading occurred, then that field will be listed here because two or more fetches might retrieve the same reading instance.

Uses the transformed version of the field names.

fetch() → *FetchResult*

Fetch new data from the source (generally the web).

static name() → str

The dataset name, which is used in the UI and for things like file and table names.

property transformer: *Transformer*

The transformer to be applied to each row that is fetched from the data source before it is stored.

Module contents

datasets - a collection of curated, built-in datasets

Submodules

mtdata.backward module

mtdata.backward.**read_backward**(*file: BinaryIO*) → Iterable[str]

Read the lines of a text file (opened in binary mode), but backward, from the last line to the first line.

```
>>> from io import BytesIO
>>> list(read_backward(BytesIO(b'a\nb')))
['b', 'a\n']
>>> list(read_backward(BytesIO(b'a\nb\n')))
['b\n', 'a\n']
>>> list(read_backward(BytesIO(b'a\n\n')))
['\n', 'a\n']
>>> list(read_backward(BytesIO(b'\n\n')))
['\n', '\n']
>>> list(read_backward(BytesIO(b'\n')))
['\n']
>>> list(read_backward(BytesIO(b'')))
[]
```

mtdata.dataset module

class mtdata.dataset.**Dataset**

Bases: ABC

A single collection of data, represented as a table.

abstract property dedup_facets: Iterable[str]

Fields used to determine which rows should be compared for de-duplication.

For example, if the data are sensor readings for various locations, the field that indicates the location would be listed here. That way, we don't drop a new reading from a different location just because it occurred at the same time as a reading from a different location.

Uses the transformed version of the field names.

abstract property dedup_fields: Iterable[str]

Fields used to compare rows for de-duplication. This is likely to be some kind of timestamp, but that depends on the kind of data.

For example, if the data are sensor readings and each row is timestamped based on when the reading occurred, then that field will be listed here because two or more fetches might retrieve the same reading instance.

Uses the transformed version of the field names.

abstract fetch() → *FetchResult*

Fetch new data from the source (generally the web).

abstract static name() → str

The dataset name, which is used in the UI and for things like file and table names.

abstract property transformer: *Transformer*

The transformer to be applied to each row that is fetched from the data source before it is stored.

class mtdata.dataset.**FetchResult**(*success: bool*, *message: str*, *data: Iterable[Dict[str, Any]]*)

Bases: tuple

The result of a completed fetch operation. If the operation was successful (data were acquired, or there were no new data available) then `success` should be `True`, `False` otherwise.

Upon success, it is reasonable for the `message` to be the empty string. However, if the operation failed, then the `message` field should contain some kind of explanation suitable for presentation to the user and inclusion in log files.

If the fetch was successful, then `data` should contain the rows that were acquired from the data source. It may be empty if there were no rows available (this will depend on the source). It should also be empty on failure.

property data

Alias for field number 2

property message

Alias for field number 1

property success

Alias for field number 0

mtdata.fields module

mtdata.fields.**prune_fields**(*row: Dict[str, Any]*, *keys: Iterable[str]*) → Dict[str, Any]

Remove fields from the row that are not included in the iterable of keys provided. The row is mutated, but also returned to the caller.

```
>>> prune_fields({'a': 1, 'b': 2}, ['b'])
{'b': 2}
```

mtdata.fields.**rename_fields**(*row: Dict[str, Any]*, *mapping: Dict[str, str]*) → Dict[str, Any]

Rename the fields of the given row using the name mapping provided. The row is mutated, but also returned to the caller.

```
>>> rename_fields({'A': 1, 'b': 2}, {'A': 'a', 'b': 'b'})
{'a': 1, 'b': 2}
```

mtdata.manifest module

mtdata.manifest.**get_dataset**(*name: str*) → Optional[Type[*Dataset*]]

Get the dataset class with the given name, or `None` if there is no dataset implementation in the manifest with that name.

mtdata.manifest.**get_store**(*name: str*) → Optional[Type[*Storage*]]

Get the store class with the given name, or `None` if there is no store implementation in the manifest with that name.

mtdata.parameters module

class mtdata.parameters.**Parameters**(*datasets: Tuple[str]*, *list_datasets: bool*, *list_stores: bool*, *namespace: str*, *stores: Tuple[str]*)

Bases: `tuple`

Parameters supported by the CLI.

property datasets

Alias for field number 0

property list_datasets

Alias for field number 1

property list_stores

Alias for field number 2

property namespace

Alias for field number 3

property stores

Alias for field number 4

mtdata.parameters.**comma_tuple**(*arg: str*) → Tuple[str, ...]

A "type" that can be used with `ArgumentParser` to split a comma-delimited list of values into an actual list.

TODO: Add a "type" to this that converts the values

mtdata.parameters.**parse_parameters**(*args: List[str]*) → *Parameters*

Turn a list of command line arguments into a `Parameters` object.

mtdata.registry module

mtdata.row module

mtdata.storage module

class mtdata.storage.**CSVBasic**(*namespace: str*)

Bases: *Storage*

A minimal CSV implementation that uses a *DictWriter* to write rows to the indicated file.

append(*name: str*, *data: Iterable[Dict[str, Any]]*, *dedup_facets: Iterable[str]*, *dedup_fields: Iterable[str]*) → *StoreResult*

Append some number of rows to the data currently stored. The existing data should remain untouched and the new data should, where it makes sense, be stored "after" the existing data.

The `name` is the identifier associated with the dataset being stored and should be used to construct any files or tables required by the storage implementation.

If `dedup_facets` and `dedup_fields` are non-empty, then de-duplication must occur before the new data are stored. See the documentation for `Dataset` for an explanation of these fields.

load(*name: str*) → Iterable[Dict[str, Any]]

Read in all data and return it as an iterable of rows. The implementation may choose to read all rows into memory or stream them through an iterator.

The `name` is the identifier associated with the dataset being stored and should be used to construct any files or tables required by the storage implementation.

load_backward(*name: str*) → Iterable[Dict[str, Any]]

Load the data in reverse order. Used for de-duplication.

static name() → str

A human-readable name for the storage implementation. Intended for use in the UI.

By convention, this should be the class name, converted to kabob case. So a storage class called `FancyDatabase` would be named "fancy-database".

name_to_path(*name: str*) → str

Name to path conversion that assumes the file extension.

replace(*name: str*, *data: Iterable[Dict[str, Any]]*) → *StoreResult*

Delete all data currently stored and replace it with the given rows.

The `name` is the identifier associated with the dataset being stored and should be used to construct any files or tables required by the storage implementation.

class mtdata.storage.**JsonLines**(*namespace: str*)

Bases: *Storage*

A storage implementation that writes each row as a single, JSON-formatted line in a file. This allows the data to be "streamed" back without reading in the entire file. It also allows efficient append operations since the old data needn't be loaded in order to add more.

Data are stored and retrieved in the order they are appended or replaced. Therefore, as long as data are always added in chronological order, they will remain in that order.

append(*name: str*, *data: Iterable[Dict[str, Any]]*, *dedup_facets: Iterable[str]*, *dedup_fields: Iterable[str]*) → *StoreResult*

Append some number of rows to the data currently stored. The existing data should remain untouched and the new data should, where it makes sense, be stored "after" the existing data.

The `name` is the identifier associated with the dataset being stored and should be used to construct any files or tables required by the storage implementation.

If `dedup_facets` and `dedup_fields` are non-empty, then de-duplication must occur before the new data are stored. See the documentation for `Dataset` for an explanation of these fields.

load(*name: str*) → Iterable[Dict[str, Any]]

Read in all data and return it as an iterable of rows. The implementation may choose to read all rows into memory or stream them through an iterator.

The `name` is the identifier associated with the dataset being stored and should be used to construct any files or tables required by the storage implementation.

load_backward(*name: str*) → Iterable[Dict[str, Any]]

Load data from the store in reverse order. In other words, the first row returned is the row that was most recently added to the store, and so on.

TODO: Consider making this abstract on the base class

static name() → str

A human-readable name for the storage implementation. Intended for use in the UI.

By convention, this should be the class name, converted to kabob case. So a storage class called `FancyDatabase` would be named "fancy-database".

name_to_path(*name: str*) → str

Convert a name to a file path with the correct extension.

replace(*name: str*, *data: Iterable[Dict[str, Any]]*) → *StoreResult*

Delete all data currently stored and replace it with the given rows.

The `name` is the identifier associated with the dataset being stored and should be used to construct any files or tables required by the storage implementation.

class mtdata.storage.**Storage**(*namespace: str*)

Bases: `ABC`

A generic storage manager that can handle writing data to a file or other persistence mechanism.

abstract append(*name: str*, *data: Iterable[Dict[str, Any]]*, *dedup_facets: Iterable[str]*, *dedup_fields: Iterable[str]*) → *StoreResult*

Append some number of rows to the data currently stored. The existing data should remain untouched and the new data should, where it makes sense, be stored "after" the existing data.

The `name` is the identifier associated with the dataset being stored and should be used to construct any files or tables required by the storage implementation.

If `dedup_facets` and `dedup_fields` are non-empty, then de-duplication must occur before the new data are stored. See the documentation for `Dataset` for an explanation of these fields.

static dedup(*existing_data: Iterable[Dict[str, Any]]*, *new_data: Iterable[Dict[str, Any]]*, *dedup_facets: Iterable[str] = ()*, *dedup_fields: Iterable[str] = ()*) → Iterable[Dict[str, Any]]

A helper function to de-duplicate data based on the given facets and fields. This algorithm won't work for every possible case, but it ought to cover the most common situations nicely.

Important: the `existing_data` iterable MUST be in reverse- chronological order. In other words, the first element of this iterable must be the most recent row added to the store.

If `dedup_facets` are provided, then for each new row, search backward through the existing data to find the most recent row that matches on those facets, then compare based on the `dedup_fields`. If they match, then the row will not be included in the returned iterable.

If there are no `dedup_facets`, but there are `dedup_fields`, then grab the most recent row from the stored data and compare it against each row of new data. If any of the new rows match, then drop that row and all rows that occurred before it, and add the remaining rows to the returned iterable.

If both dedup parameters are empty, then the new data are passed through unfiltered.

```
>>> list(Storage.dedup(
...   reversed([{'a': 1}, {'a': 2}]),
...   [{'a': 3}], [], ['a']))
[{'a': 3}]
>>> list(Storage.dedup(
...   reversed([{'a': 1}, {'a': 2}]),
...   [{'a': 2}, {'a': 3}], [], ['a']))
[{'a': 3}]
```

get_path(*name: str*, *extension: str*) → str

A helper for implementations that use the filesystem. Returns a path to a file with the given name and extension, located in a directory determined by the `namespace` property.

abstract load(*name: str*) → Iterable[Dict[str, Any]]

Read in all data and return it as an iterable of rows. The implementation may choose to read all rows into memory or stream them through an iterator.

The `name` is the identifier associated with the dataset being stored and should be used to construct any files or tables required by the storage implementation.

abstract static name() → str

A human-readable name for the storage implementation. Intended for use in the UI.

By convention, this should be the class name, converted to kabob case. So a storage class called `FancyDatabase` would be named "fancy-database".

property namespace: str

The namespace is tied to the instance of the running software and should be used to construct storage paths.

abstract replace(*name: str*, *data: Iterable[Dict[str, Any]]*) → *StoreResult*

Delete all data currently stored and replace it with the given rows.

The `name` is the identifier associated with the dataset being stored and should be used to construct any files or tables required by the storage implementation.

class mtdata.storage.**StoreResult**(*success: bool*, *message: str*)

Bases: `tuple`

The result of a write operation on a store.

TODO: We should wrap read operation results as well

property message

Alias for field number 1

property success

Alias for field number 0

mtdata.transformer module

class mtdata.transformer.**Transformer**(*fields: Iterable[Union[Tuple[str, str, Callable[[Any], Any]], Tuple[str, str]]] = ()*)

Bases: object

A transformation that can be applied to a single dataset row represented as a dictionary. The transformation can update field names and make arbitrary changes to field data.

When the transformation is applied, the following will happen:

1. Any fields not included in the transformation will be pruned
2. Fields that have old names specified will be renamed
3. Values will be updated for fields that have update functions

The row will be transformed in-place but also returned to the caller.

```
>>> t = Transformer()
>>> t.add_field('a', 'A')
>>> t.add_field('b', 'B', lambda x: x.lower())
>>> t({'A': 'XYZ', 'B': 'XYZ'})
{'a': 'XYZ', 'b': 'xyz'}
```

add_field(*name: str, old_name: ~typing.Optional[str] = None, updater: ~typing.Callable[[~typing.Any], ~typing.Any] = <function Transformer.<lambda>>*) → None

Add a field to the transformation.

```
>>> t = Transformer()
>>> t.add_field('a', 'A', lambda x: x.lower())
>>> t._name_mapping
{'A': 'a'}
>>> t._update_functions['a']('ABC')
'abc'
```

Module contents

mtdata - a tool for extracting and curating public data

1.2 Adding a Dataset

A dataset is implemented as a sub-class of the *Dataset* abstract class in the *mtdata.datasets* module. Once the dataset is implemented, add the class to the list in *mtdata.manifest*. Once this is done, the new dataset will run by default when the *mtdata* module is run.

It may be easiest to adapt an existing dataset, e.g. *mtdata.datasets.air_quality*. See *mtdata.dataset* for specific documentation on *mtdata.dataset.Dataset* methods.

1.3 Adding a Store

CHAPTER

TWO

INSTALLATION

CHAPTER

THREE

CONCEPTS

A **dataset** is a collection of related data, usually a single table. Each dataset knows how to download updates to the data, how to transform fields names and values, and which fields are useful for de-duplication.

A **store** is a means of persisting acquired data. For example, a particular store might know how to talk to a database to save data collected by datasets as database tables.

CHAPTER

FOUR

USAGE

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

m

INDEX

N

P

R

S

T