MDSynthesis Documentation

Release .0

David Dotson

Jun 29, 2018

Contents

Although the raw data for any study involving molecular dynamics simulations are the full trajectories themselves, often we are most interested in lower-dimensional measures of what is happening. These measures may be as simple as the distance between two specific atoms, or as complex as the percentage of contacts relative to some native structure. Some measures may even be comparisons of two or more trajectories against each other. In any case, it may be time-consuming to obtain these lower-dimensional intermediate data, and so it is useful to store them.

CHAPTER 1

Stay organized

MDSynthesis is designed to perform the logistics of medium-to-large-scale analysis of many trajectories, individually or as entire groups. It is intended to allow the scientist to operate at a high level when working with the data, while letting MDSynthesis handle the details of storing and recalling this data.

In other words, MDSynthesis lets the computer do the boring work of keeping track of where things are and how they are stored.

1.1 Efficiently store intermediate data from individual simulations for easy recall

For a given simulation trajectory, MDSynthesis gives an interface (the *Sim* object) to the simulation data itself through MDAnalysis. Data structures generated from raw trajectories (pandas objects, numpy arrays, or any pure python structure) can then be stored and easily recalled later. Under the hood, datasets are stored in the efficient HDF5 format when possible.

1.2 Collect aggregated data and keep track of it, too

Sim objects can be gathered into arbitrary collections with *Group* objects. Groups can store datasets obtained from these collections, and can even contain other Groups as members.

1.3 Query for simulation results instead of manually hunting for them

Note: This feature is planned, but not yet present in the codebase.

Sim and *Group* objects persistently store their data to disk automatically, but it can be tedious to navigate around the filesystem to recall them later. The *Coordinator* object gives a single interface for querying all *Sim* and *Group* objects it is made aware of, allowing retrieval of specific datasets with a single line of code.

CHAPTER 2

Getting MDSynthesis

We have yet to make an official release, but you can get the current state of the codebase from the master branch on GitHub.

See the *installation instructions* to set it up.

CHAPTER 3

Dependencies

- MDAnalysis: 0.9.1 or higher
- pandas: 0.16.1 or higher
- PyTables: 3.2.0 or higher
- h5py: 2.5.0 or higher
- scandir: 1.0 or higher

CHAPTER 4

Contributing

This project is still under heavy development, and there are certainly rough edges and bugs. Issues and pull requests welcome!

4.1 Documentation

4.1.1 Installation

There are no official releases of MDSynthesis yet, but the master branch on GitHub gives the most current state of the package.

First install the dependencies. Since MDSynthesis uses HDF5 as the file format of choice for persistence, you will need to install the libraries either using your package manager or manually.

On Ubuntu 14.04 this will be

```
apt-get install libhdf5-serial-1.8.4 libhdf5-serial-dev
```

and on Arch Linux

```
pacman -S hdf5
```

PyTables can be particularly picky, and it often fails to obtain its own dependencies. It is best to first install PyTables' dependencies explicitly

```
pip install numpy numexpr Cython
```

Then install PyTables and everything else

```
pip install tables
pip install pandas h5py MDAnalysis scandir
```

Then clone the repository and switch to the master branch

```
git clone git@github.com:dotsdl/MDSynthesis.git
cd MDSynthesis
git checkout master
```

Installation of the packages is as simple as

```
python setup.py build
python setup.py install
```

This installs MDSynthesis in the system wide python directory; this may require administrative privileges.

It is also possible to use `--prefix`, `--home`, or `--user` options for setup.py to install in a different (probably your private) python directory hierarchy. `python setup.py install --help` should show you your options.

4.1.2 Datasets and Containers

MDSynthesis is not an analysis code. On its own, it does not produce output data given raw simulation data as input. Its scope is limited to the boring but tedious task of data management and storage. It is intended to bring value to analysis results by making them easily accessible now and later.

As such, the basic functionality of MDSynthesis is condensed into only two objects, sometimes referred to as *Containers* in the documentation. These are the *Sim* and *Group* objects.

In brief, a **Sim** is designed to manage and give access to the data corresponding to a single simulation (the raw trajectory(s), as well as analysis results); a **Group** gives access to any number of **Sim** or **Group** objects it has as members (including perhaps itself), and can store analysis results that pertain to these members collectively. Both types of Container store their underlying data persistently to disk on the fly. The file locking needed for each transaction is handled automatically, so more than one python process can be working with any number of instances of the same Container at the same time.

> **Warning:** File locking is generally process safe, but not thread safe. Don't use multithreading and try to modify Container elements at the same time. Multiprocessing, however, should work just fine.

Persistence as a feature

Containers store their data as directory structures in the file system. Generating a new **Sim**, for example, with the following

```
>>> # python session 1
>>> import mdsynthesis as mds
>>> s = mds.Sim('marklar')
```

creates a directory called `marklar` in the current working directory. It contains a single file at the moment

```
> # shell
> ls marklar
Sim.2b4b5800-48a7-4814-ba6d-1e631a09a199.h5
```

The name of this file includes the type of Container (**Sim**) it corresponds to, as well as the `uuid` of the Container, which is its unique identifier. This is the state file containing all the information needed to regenerate an identical instance of this **Sim**. In fact, we can open a separate python session (go ahead!) and regenerate this **Sim** immediately there

```
>>> # python session 2
>>> import mdsynthesis as mds
>>> s = mds.Sim('marklar')
```

Making a modification to the **Sim** in one session, perhaps by adding a tag, will be reflected in the **Sim** in the other session

```
>>> # python session 1
>>> s.tags.add('TIP4P')

>>> # python session 2
>>> s.tags
<Tags(['TIP4P'])>
```

This is because both objects pull their identifying information from the same file on disk; they store almost nothing in memory.

Note: The `uuid` of the **Sim** in this example will certainly differ from any **Sims** you generate. This is used to differentiate **Sims** from each other. Unexpected and broken behavior will result from changing the names of state files!

Storing arbitrary datasets

More on things like tags later, but we really care about storing (potentially large and time consuming to produce) datasets. Using our **Sim** `marklar` as the example here, say we have generated a numpy array of dimension (10^6, 3) that gives the minimum distance between the sidechains of three residues with those of a fourth for each frame in a trajectory

```
>>> a.shape
(1000000, 3)
```

We can store this easily

```
>>> s.data.add('distances', a)
>>> s.data
<Data(['distances'])>
```

and recall it

```
>>> s.data['distances'].shape
(1000000, 3)
```

Looking at the contents of the directory `marklar`, we see it has a new subdirectory corresponding to the name of our stored dataset

```
> # shell
> ls marklar
distances  Sim.h5
```

which has its own contents

```
> ls marklar/distances
npData.h5
```

This is the data we stored, serialized to disk in the efficient HDF5 data format. Containers will also store pandas objects using this format. For other data structures, the Container will pickle them if it can.

Datasets can be nested however you like. For example, say we had several pandas **DataFrames** each giving the distance with time of each cation in the simulation with respect to some residue of interest on our protein. We could just as well make it clear to ourselves that these are all similar datasets by grouping them together

```
>>> s.data.add('cations/residue1', df1)
>>> s.data.add('cations/residue2', df2)
>>> # we can also use setitem syntax
>>> s.data['cations/residue3'] = df3
>>> s.data
<Data(['cations/residue1', 'cations/residue2', cations/residue3',
       'distances'])>
```

and their locations in the filesystem reflect this structure.

Minimal blobs

Individual datasets get their own place in the filesystem instead of all being shoved into a single file on disk. This is by design, as it generally means better performance since this means less waiting for file locks to release from other Container instances. Also, it gives a space to put other files related to the dataset itself, such as figures produced from it.

You can get the location on disk of a dataset with

```
>>> s.data.locate('cations/residue1')
'/home/bob/marklar/cations/residue1'
```

which is particularly useful for outputting figures.

Another advantage of organizing Containers at the filesystem level is that datasets can be handled at the filesystem level. Removing a dataset with a

```
> # shell
> rm -r marklar/cations/residue2
```

is immediately reflected by the Container

```
>>> s.data
<Data(['cations/residue1', 'cations/residue3', 'distances'])>
```

Datasets can likewise be moved within the Container's directory tree and they will still be found, with names matching their location relative to the state file.

Reference: Data

The class *mdsynthesis.core.aggregators.Data* is the interface used by Containers to access their stored datasets. It is not intended to be used on its own, but is shown here to give a detailed view of its methods.

class mdsynthesis.core.aggregators.**Data**(*container*, *containerfile*, *logger*)

Interface to stored data.

add(*handle*, **args*, ***kwargs*)

Store data in Container.

A data instance can be a pandas object (Series, DataFrame, Panel), a numpy array, or a pickleable python object. If the dataset doesn't exist, it is added. If a dataset already exists for the given handle, it is replaced.

Arguments

handle name given to data; needed for retrieval

data data structure to store

append(*handle*, **args*, ***kwargs*)
Append rows to an existing dataset.

The object must be of the same pandas class (Series, DataFrame, Panel) as the existing dataset, and it must have exactly the same columns (names included).

Arguments

handle name of data to append to

data data to append

locate(*handle*)
Get directory location for a stored dataset.

Arguments

handle name of data to retrieve location of

Returns

datadir absolute path to directory containing stored data

make_filepath(*handle*, *filename*)
Return a full path for a file stored in a data directory, whether the file exists or not.

This is useful if preparing plots or other files derived from the dataset, since these can be stored with the data in its own directory. This method does the small but annoying work of generating a full path for the file.

This method doesn't care whether or not the path exists; it simply returns the path it's asked to build.

Arguments

handle name of dataset file corresponds to

filename filename of file

Returns

filepath absolute path for file

remove(*handle*, ***kwargs*)
Remove a dataset, or some subset of a dataset.

Note: in the case the whole dataset is removed, the directory containing the dataset file (`Data.h5`) will NOT be removed if it still contains file(s) after the removal of the dataset file.

For pandas objects (Series, DataFrame, or Panel) subsets of the whole dataset can be removed using keywords such as *start* and *stop* for ranges of rows, and *columns* for selected columns.

Arguments

handle name of dataset to delete

Keywords

where conditions for what rows/columns to remove

start row number to start selection

stop row number to stop selection

columns columns to remove

retrieve(*handle*, **args*, ***kwargs*)

Retrieve stored data.

The stored data structure is read from disk and returned.

If dataset doesn't exist, `None` is returned.

For pandas objects (Series, DataFrame, or Panel) subsets of the whole dataset can be returned using keywords such as *start* and *stop* for ranges of rows, and *columns* for selected columns.

Also for pandas objects, the *where* keyword takes a string as input and can be used to filter out rows and columns without loading the full object into memory. For example, given a DataFrame with handle 'mydata' with columns (A, B, C, D), one could return all rows for columns A and C for which column D is greater than .3 with:

```
retrieve('mydata', where='columns=[A,C] & D > .3')
```

Or, if we wanted all rows with index = 3 (there could be more than one):

```
retrieve('mydata', where='index = 3')
```

See :meth:pandas.HDFStore.select() for more information.

Arguments

handle name of data to retrieve

Keywords

where conditions for what rows/columns to return

start row number to start selection

stop row number to stop selection

columns list of columns to return; all columns returned by default

iterator if True, return an iterator [`False`]

chunksize number of rows to include in iteration; implies `iterator=True`

Returns

data stored data; `None` if nonexistent

4.1.3 Using Sims to dissect trajectories

Sim objects are designed to store datasets that are obtained from a single simulation, and they give a direct interface to trajectory data by way of the MDAnalysis **Universe** object.

To generate a **Sim** from scratch, we need only give it a name. This will be used to distinguish the **Sim** from others, though it need not be unique. We can also give it a topology and/or trajectory files as we would to an MDAnalysis **Universe**

```
>>> from mdsynthesis import Sim
>>> s = Sim('scruffy', universe=['path/to/topology', 'path/to/trajectory'])
```

This will create a directory `scruffy` that contains a single file (`Sim.<uuid>.h5`). That file is a persistent representation of the **Sim** on disk. We can access trajectory data by way of

```
>>> s.universe
<Universe with 47681 atoms>
```

The **Sim** can also store selections by giving the usual inputs to `Universe.selectAtoms`

```
>>> s.selections.add('backbone', 'name CA', 'name N', 'name C')
```

And the **AtomGroup** can be conveniently obtained with

```
>>> s.selections['backbone']
<AtomGroup with 642 atoms>
```

Note: Only selection strings are stored, not the resulting atoms of those selections. This means that if the topology of the **Universe** is replaced or altered, the **AtomGroup** returned by a particular selection may change.

Multiple Universes

Often it is necessary to post-process a simulation trajectory to get it into a useful form for analysis. This may involve coordinate transformations that center on a particular set of atoms or fit to a structure, removal of water, skipping of frames, etc. This can mean that for a given simulation multiple versions of the raw trajectory may be needed.

For this reason, a **Sim** can store multiple **Universe** definitions. To add a definition, we need a topology and a trajectory file

```
>>> s.universes.add('anotherU', 'path/to/topology', 'path/to/trajectory')
>>> s.universes
<Universes(['anotherU', 'main'])>
```

and we can make this the active **Universe** with

```
>>> s.universes['anotherU']
>>> s
<Sim: 'scruffy' | active universe: 'anotherU'>
```

Only a single **Universe** may be active at a time. Atom selections that are stored correspond to the currently active **Universe**, since different selection strings may be required to achieve the same selection under a different **Universe** definition. For convenience, we can copy the selections corresponding to another **Universe** to the active **Universe** with

```
>>> s.selections.copy('main')
```

Need two **Universe** definitions to be active at the same time? Re-generate a second **Sim** instance from its representation on disk and activate the desired **Universe**.

Resnums can also be stored

Depending on the simulation package used, it may not be possible to have the resids of the protein match those given in, say, the canonical PDB structure. This can make selections by resid cumbersome at best. For this reason, residues can also be assigned resnums.

For example, say the resids for the protein in our **Universe** range from 1 to 214, but they should actually go from 10 to 223. If we can't change the topology to reflect this, we could set the resnums for these residues to the canonical values

```
>>> prot = s.universe.selectAtoms('protein')
>>> prot.residues.set_resnum(prot.residues.resids() + 9)
>>> prot.residues.resnums()
array([ 10,  11,  12,  13,  14,  15,  16,  17,  18,  19,  20,  21,  22,
        23,  24,  25,  26,  27,  28,  29,  30,  31,  32,  33,  34,  35,
        36,  37,  38,  39,  40,  41,  42,  43,  44,  45,  46,  47,  48,
        49,  50,  51,  52,  53,  54,  55,  56,  57,  58,  59,  60,  61,
        62,  63,  64,  65,  66,  67,  68,  69,  70,  71,  72,  73,  74,
        75,  76,  77,  78,  79,  80,  81,  82,  83,  84,  85,  86,  87,
        88,  89,  90,  91,  92,  93,  94,  95,  96,  97,  98,  99, 100,
       101, 102, 103, 104, 105, 106, 107, 108, 109, 110, 111, 112, 113,
       114, 115, 116, 117, 118, 119, 120, 121, 122, 123, 124, 125, 126,
       127, 128, 129, 130, 131, 132, 133, 134, 135, 136, 137, 138, 139,
       140, 141, 142, 143, 144, 145, 146, 147, 148, 149, 150, 151, 152,
       153, 154, 155, 156, 157, 158, 159, 160, 161, 162, 163, 164, 165,
       166, 167, 168, 169, 170, 171, 172, 173, 174, 175, 176, 177, 178,
       179, 180, 181, 182, 183, 184, 185, 186, 187, 188, 189, 190, 191,
       192, 193, 194, 195, 196, 197, 198, 199, 200, 201, 202, 203, 204,
       205, 206, 207, 208, 209, 210, 211, 212, 213, 214, 215, 216, 217,
       218, 219, 220, 221, 222, 223])
```

We can now select residue 95 from the PDB structure with

```
>>> s.universe.selectAtoms('protein and resnum 95')
```

and we might save selections using resnums as well. However, resnums aren't stored in the topology, so to avoid having to reset resnums manually each time we load the **Universe**, we can just store the resnum definition with

```
>>> s.universes.resnums('main', s.universe.residues.resnums())
```

and the resnum definition will be applied to the **Universe** both now and every time it is activated.

Reference: Sim

class mdsynthesis.**Sim**(*sim*, *universe=None*, *uname='main'*, *location='.'*, *coordinator=None*, *categories=None*, *tags=None*)

The Sim object is an interface to data for single simulations.

Generate a new or regenerate an existing (on disk) Sim object.

Required arguments

sim if generating a new Sim, the desired name to give it; if regenerating an existing Sim, string giving the path to the directory containing the Sim object's state file

Optional arguments when generating a new Sim

uname desired name to associate with universe; this universe will be made the default (can always be changed later)

universe arguments usually given to an MDAnalysis Universe that defines the topology and trajectory of the atoms

location directory to place Sim object; default is the current directory

coordinator directory of the Coordinator to associate with the Sim; if the Coordinator does not exist, it is created; if `None`, the Sim will not associate with any Coordinator

categories dictionary with user-defined keys and values; used to give Sims distinguishing characteristics

tags list with user-defined values; like categories, but useful for adding many distinguishing descriptors

***Note*: optional arguments are ignored when regenerating an existing** Sim

basedir
Absolute path to the Container's base directory.

This is a convenience property; the same result can be obtained by joining *:attr:location* and *:attr:name*.

categories
The categories of the Container.

Categories are user-added key-value pairs that can be used to and distinguish Containers from one another through Coordinator or Group queries. They can also be useful as flags for external code to determine how to handle the Container.

containertype
The type of the Container.

coordinators
The locations of the associated Coordinators.

Change this to associate the Container with an existing or new Coordinator(s).

data
The data of the Container.

Data are user-generated pandas objects (e.g. Series, DataFrames), numpy arrays, or any pickleable python object that are stored in the Container for easy recall later. Each data instance is given its own directory in the Container's tree.

location
The location of the Container.

Setting the location to a new path physically moves the Container to the given location. This only works if the new location is an empty or nonexistent directory.

name
The name of the Container.

The name of a Container need not be unique with respect to other Containers, but is used as part of Container's displayed representation.

selections
Stored atom selections for the active universe.

Useful atom selections can be stored for the active universe and recalled later. Selections are stored separately for each defined universe, since the same selection may require a different selection string for different universes.

tags
The tags of the Container.

Tags are user-added strings that can be used to and distinguish Containers from one another through Coordinator or Group queries. They can also be useful as flags for external code to determine how to handle the Container.

universe
The active universe of the Sim.

Universes are interfaces to raw simulation data. The Sim can store multiple universe definitions corresponding to different versions of the same simulation output (e.g. post-processed trajectories derived from

the same raw trajectory). The Sim has at most one universe definition that is "active" at one time, with stored selections for this universe directly available via `Sim.selections`.

To have more than one universe available as "active" at the same time, generate as many instances of the Sim object from the same statefile on disk as needed, and make a universe active for each one.

universes
Manage the defined universes of the Sim.

Universes are interfaces to raw simulation data. The Sim can store multiple universe definitions corresponding to different versions of the same simulation output (e.g. post-processed trajectories derived from the same raw trajectory). The Sim has at most one universe definition that is "active" at one time, with stored selections for this universe directly available via `Sim.selections`.

The Sim can also store a preference for a "default" universe, which is activated on a call to `Sim.universe` when no other universe is active.

uuid
Get Container uuid.

A Container's uuid is used by other Containers to identify it. The uuid is given in the Container's state file name for fast filesystem searching. For example, a Sim object with state file:

```
'Sim.7dd9305a-d7d9-4a7b-b513-adf5f4205e09.h5'
```

has uuid:

```
'7dd9305a-d7d9-4a7b-b513-adf5f4205e09'
```

Changing this string will alter the Container's uuid. This is not generally recommended.

Returns

uuid unique identifier string for this Container

Reference: Universes

The class *mdsynthesis.core.aggregators.Universes* is the interface used by a **Sim** to manage **Universe** definitions. It is not intended to be used on its own, but is shown here to give a detailed view of its methods.

class mdsynthesis.core.aggregators.**Universes**(*container*, *containerfile*, *logger*)
Interface to universes.

activate(*handle=None*)
Make the selected universe active.

Only one universe definition can be active in a Sim at one time. The active universe can be accessed from `Sim.universe`. Stored selections for the active universe can be accessed as items in `Sim.selections`.

If no handle given, the default universe is loaded.

If a resnum definition exists for the universe, it is applied.

Arguments

handle given name for selecting the universe; if `None`, default universe selected

add(*handle*, *topology*, **trajectory*)
Add a universe definition to the Sim object.

A universe is an MDAnalysis object that gives access to the details of a simulation trajectory. A Sim object can contain multiple universe definitions (topology and trajectory pairs), since it is often convenient to have different post-processed versions of the same raw trajectory.

Using an existing universe handle will replace the topology and trajectory for that definition; selections for that universe will be retained.

If there is no current default universe, then the added universe will become the default.

Arguments

handle given name for selecting the universe

topology path to the topology file

trajectory path to the trajectory file; multiple files may be given and these will be used in order as frames for the trajectory

`current``()`

Return the name of the currently active universe.

Returns

handle name of currently active universe

`deactivate``()`

Deactivate the current universe.

Deactivating the current universe may be necessary to conserve memory, since the universe can then be garbage collected.

`default``(`*handle=None*`)`

Mark the selected universe as the default, or get the default universe.

The default universe is loaded on calls to `Sim.universe` or `Sim.selections` when no other universe is attached.

If no handle given, returns the current default universe.

Arguments

handle given name for selecting the universe; if `None`, default universe is unchanged

Returns

default handle of the default universe

`define``(`*handle*, *pathtype='abspath'*`)`

Get the stored path to the topology and trajectory used for the specified universe.

***Note*: Does no checking as to whether these paths are valid. To** check this, try activating the universe.

Arguments

handle name of universe to get definition for

Keywords

pathtype type of path to return; 'abspath' gives an absolute path, 'relCont' gives a path relative to the Sim's state file

Returns

topology path to the topology file

trajectory list of paths to trajectory files

remove (**handle*)
Remove a universe definition.

Also removes any selections associated with the universe.

Arguments

handle name of universe(s) to delete

resnums (*handle*, *resnums*)
Define resnums for the given universe.

Resnums are useful for referring to residues by their canonical resid, for instance that stored in the PDB. By giving a resnum definition for the universe, this definition will be applied to the universe on activation.

Will overwrite existing resnum definition if it exists.

Arguments

handle name of universe to apply resnums to

resnums list giving the resnum for each residue in the topology, in atom index order; giving `None` will delete resnum definition

Reference: Selections

The class *mdsynthesis.core.aggregators.Selections* is the interface used by a **Sim** to access its stored selections. It is not intended to be used on its own, but is shown here to give a detailed view of its methods.

class mdsynthesis.core.aggregators.**Selections**(*container*, *containerfile*, *logger*)
Selection manager for Sims.

Selections are accessible as items using their handles. Each time they are called, they are regenerated from the universe that is currently active. In this way, changes in the universe topology are reflected in the selections.

add(*handle*, **selection*)
Add an atom selection for the attached universe.

AtomGroups are needed to obtain useful information from raw coordinate data. It is useful to store AtomGroup selections for later use, since they can be complex and atom order may matter.

If a selection with the given *handle* already exists, it is replaced.

Arguments

handle name to use for the selection

selection selection string; multiple strings may be given and their order will be preserved, which is useful for e.g. structural alignments

asAtomGroup (*handle*)
Get AtomGroup from active universe from the given named selection.

If named selection doesn't exist, `KeyError` raised.

Arguments

handle name of selection to return as an AtomGroup

Returns

AtomGroup the named selection as an AtomGroup of the active universe

copy (*universe*)
Copy defined selections of another universe to the active universe.

Arguments

universe name of universe definition to copy selections from

define(*handle*)

Get selection definition for given handle and the active universe.

If named selection doesn't exist, `KeyError` raised.

Arguments

handle name of selection to get definition of

Returns

definition list of strings defining the atom selection

keys()

Return a list of all selection handles.

remove(**handle*)

Remove an atom selection for the attached universe.

If named selection doesn't exist, `KeyError` raised.

Arguments

handle name of selection(s) to remove

4.1.4 Leveraging Groups for aggregate data

Group objects can keep track of any number of **Sim** and **Group** objects it counts as members, and it can store datasets derived from these objects. Just as a **Sim** manages data obtained from a single simulation, a **Group** is designed to manage data obtained from a collection of **Sim** or **Group** objects in aggregate.

As with a **Sim**, to generate a Group from scratch, we need only give it a name. We can also give any number of existing **Sim** or **Group** objects to add them as members

```
>>> from mdsynthesis import Group
>>> g = Group('gruffy', members=[s1, s2, s3, g4, g5])
>>> g
<Group: 'gruffy' | 5 Members: 3 Sim, 2 Group>
```

This will create a directory `gruffy` that contains a single file (`Group.<uuid>.h5`). That file is a persistent representation of the **Group** on disk. We can access its members with

```
>>> g.members
<Members(['marklar', 'scruffy', 'fluffy', 'buffy', 'gorp'])>
>>> g.members[2]
<Sim: 'fluffy'>
```

and we can slice, too

```
>>> g.members[2:]
[<Sim: 'fluffy'>, <Group: 'buffy'>, <Group: 'gorp'>]
```

Note: Members are generated from their state files on disk upon access. This means that for a **Group** with hundreds of members, there will be a delay when trying to access them all at once.

A **Group** can even be a member of itself

```
>>> g.members.add(g)
>>> g
<Group: 'gruffy' | 6 Members: 3 Sim, 3 Group>
>>> g.members[-1]
<Group: 'gruffy' | 6 Members: 3 Sim, 3 Group>
>>> g.members[-1].members[-1]
<Group: 'gruffy' | 6 Members: 3 Sim, 3 Group>
```

As a technical aside, note that a **Group** returned as a member of itself is not the same object in memory as the **Group** that returned it. They are two different instances of the same **Group**

```
>>> g2 = g.members[-1]
>>> g2 is g
False
```

But since they pull their state from the same file on disk, they will reflect the same stored information at all times

```
>>> g.tags.add('kinases')
>>> g2.tags
<Tags(['kinases'])>
```

Reference: Group

class mdsynthesis.**Group**(*group*, *members=None*, *location='.'*, *coordinator=None*, *categories=None*, *tags=None*)

The Group object is a collection of Sims and Groups.

Generate a new or regenerate an existing (on disk) Group object.

Required Arguments

group if generating a new Group, the desired name to give it; if regenerating an existing Group, string giving the path to the directory containing the Group object's state file

Optional arguments when generating a new Group

members a list of Sims and/or Groups to immediately add as members

location directory to place Group object; default is the current directory

coordinator directory of the Coordinator to associate with this object; if the Coordinator does not exist, it is created; if `None`, the Sim will not associate with any Coordinator

categories dictionary with user-defined keys and values; used to give Groups distinguishing characteristics

tags list with user-defined values; like categories, but useful for adding many distinguishing descriptors

***Note*: optional arguments are ignored when regenerating an existing** Group

basedir

Absolute path to the Container's base directory.

This is a convenience property; the same result can be obtained by joining *:attr:location* and *:attr:name*.

categories

The categories of the Container.

Categories are user-added key-value pairs that can be used to and distinguish Containers from one another through Coordinator or Group queries. They can also be useful as flags for external code to determine how to handle the Container.

containertype
The type of the Container.

coordinators
The locations of the associated Coordinators.

Change this to associate the Container with an existing or new Coordinator(s).

data
The data of the Container.

Data are user-generated pandas objects (e.g. Series, DataFrames), numpy arrays, or any pickleable python object that are stored in the Container for easy recall later. Each data instance is given its own directory in the Container's tree.

location
The location of the Container.

Setting the location to a new path physically moves the Container to the given location. This only works if the new location is an empty or nonexistent directory.

members
The members of the Group.

A Group is useful as an interface to collections of Containers, and they allow direct access to each member of that collection. Often a Group is used to store datasets derived from this collection as an aggregate.

Queries can also be made on the Group's members to return a subselection of the members based on some search criteria. This can be useful to define new Groups from members of existing ones.

name
The name of the Container.

The name of a Container need not be unique with respect to other Containers, but is used as part of Container's displayed representation.

tags
The tags of the Container.

Tags are user-added strings that can be used to and distinguish Containers from one another through Coordinator or Group queries. They can also be useful as flags for external code to determine how to handle the Container.

uuid
Get Container uuid.

A Container's uuid is used by other Containers to identify it. The uuid is given in the Container's state file name for fast filesystem searching. For example, a Sim object with state file:

```
'Sim.7dd9305a-d7d9-4a7b-b513-adf5f4205e09.h5'
```

has uuid:

```
'7dd9305a-d7d9-4a7b-b513-adf5f4205e09'
```

Changing this string will alter the Container's uuid. This is not generally recommended.

Returns

uuid unique identifier string for this Container

Reference: Members

The class *mdsynthesis.core.aggregators.Members* is the interface used by a **Group** to manage its members. It is not intended to be used on its own, but is shown here to give a detailed view of its methods.

class mdsynthesis.core.aggregators.**Members**(*container*, *containerfile*, *logger*)
Member manager for Groups.

add(**containers*)
Add any number of members to this collection.

Arguments

containers Sims and/or Groups to be added; may be a list of Sims and/or Groups; Sims or Groups can be given as either objects or paths to directories that contain object statefiles

containertypes
Return a list of member containertypes.

data
The data of the Container.

Data are user-generated pandas objects (e.g. Series, DataFrames), numpy arrays, or any pickleable python object that are stored in the Container for easy recall later. Each data instance is given its own directory in the Container's tree.

names
Return a list of member names.

Members that can't be found will have name `None`.

Returns

names list giving the name of each member, in order; members that are missing will have name `None`

remove(**members*, ***kwargs*)
Remove any number of members from the Group.

Arguments

members instances or indices of the members to remove

Keywords

all When True, remove all members [`False`]

uuids
Return a list of member uuids.

Returns

uuids list giving the uuid of each member, in order

4.1.5 Differentiating Containers

Sims and **Groups** can be used to develop "fire-and-forget" analysis routines. Large numbers of Containers can be fed to an analysis code to give that code access to all trajectory and intermediate data, with individual Containers handled according to their characteristics. To make it possible to write code that tailors its approach according to the Container it encounters, we can use tags and categories.

Tags are individual strings that describe a Container. Using our **Sim** `marklar` as an example, we can add many tags at once

```
>>> from mdsynthesis import Sim
>>> s = Sim('marklar')
>>> s.tags.add('TIP4P', 'ADK', 'kinases', 'globular', 'equilibrium')
>>> s.tags
<Tags(['ADK', 'TIP4P', 'equilibrium', 'globular', 'kinases'])>
```

They can be iterated through as well

```
>>> for tag in s.tags:
>>>     print tag
kinases
globular
ADK
TIP4P
equilibrium
```

Categories are key-value pairs of strings. They are particularly useful as switches for analysis code. For example, if we are simulating two different states of a protein (say, "open" and "closed"), we can make a category that reflects this. In this case, we categorize `marklar` as "open"

```
>>> s.categories['state'] = 'open'
>>> s.categories
<Categories({'state': 'open'})>
```

Perhaps we've written some analysis code that will take both "open" and "closed" simulation trajectories as input but needs to handle them differently. It can see what variety of **Sim** it is working with using

```
>>> s.categories['state']
'open'
```

Future: Querying

Tags and categories are two elements of Containers that will be *queryable*.

Reference: Tags

The class *mdsynthesis.core.aggregators.Tags* is the interface used by Containers to access their tags. It is not intended to be used on its own, but is shown here to give a detailed view of its methods.

class mdsynthesis.core.aggregators.**Tags**(*container*, *containerfile*, *logger*)

Interface to tags.

add(**tags*)

Add any number of tags to the Container.

Tags are individual strings that serve to differentiate Containers from one another. Sometimes preferable to categories.

Arguments

tags Tags to add. Must be convertable to strings using the str() builtin. May also be a list of tags.

remove(**tags*, ***kwargs*)
Remove tags from Container.

Any number of tags can be given as arguments, and these will be deleted.

Arguments

tags Tags to delete.

Keywords

all When True, delete all tags [`False`]

Reference: Categories

The class *mdsynthesis.core.aggregators.Categories* is the interface used by Containers to access their categories. It is not intended to be used on its own, but is shown here to give a detailed view of its methods.

class mdsynthesis.core.aggregators.**Categories**(*container*, *containerfile*, *logger*)
Interface to categories.

add(**categorydicts*, ***categories*)
Add any number of categories to the Container.

Categories are key-value pairs of strings that serve to differentiate Containers from one another. Sometimes preferable to tags.

If a given category already exists (same key), the value given will replace the value for that category.

Keywords

categorydict dict of categories to add; keys used as keys, values used as values. Both keys and values must be convertible to strings using the str() builtin.

categories Categories to add. Keyword used as key, value used as value. Both must be convertible to strings using the str() builtin.

keys()
Get category keys.

Returns

keys keys present among categories

remove(**categories*, ***kwargs*)
Remove categories from Container.

Any number of categories (keys) can be given as arguments, and these keys (with their values) will be deleted.

Arguments

categories Categories to delete.

Keywords

all When True, delete all categories [`False`]

values()
Get category values.

Returns

values values present among categories

4.1.6 Query and high-level control with Coordinators

Because **Sims** and **Groups** store their information neatly in their state files, this data can be aggregated and queried. This allows whole selections of Containers to be manipulated without needing to hunt them down in the filesystem. The **Coordinator** object gives an interface for doing this. **Sims** and **Groups** that are associated with a given **Coordinator** will report changes to their state files as they are made, giving the **Coordinator** a thin copy of all Containers it is made aware of.

This feature is not yet implemented.

4.2 Misc

4.2.1 Frequently Asked Questions

1. Why PyTables?

   PyTables is a (fantastic) interface to the hdf5 data format. Although not itself a relational database, MDSynthesis uses PyTables for building and managing the persistent state files on disk for **Sim**, **Group**, and **Coordinator** objects. This was chosen over a traditional RDBS because we wanted MDSynthesis to be serverless, and SQLite was not ideal because its file locking mechanisms are known to be unreliable on a network file system (NFS).

Index

R

S

T

U

V