datreant.data Documentation

Release 0.7.1

David Dotson

May 07, 2018

User Documentation

1 Getting datreant.data 3

2 Contributing 5

This datreant submodule adds a convenience interface for numpy and pandas data storage and retrieval using HDF5 within a Treant's directory structure. It provides the `data` limb for Treants, Trees, Bundles, and Views from datreant.core.

> **Warning:** This module is **experimental**. It is not API stable, and has many rough edges and limitations. It is, however, usable.

CHAPTER 1

Getting datreant.data

See the *installation instructions* for installation details. The package itself is pure Python, but it is dependent on HDF5 libraries and the Python interfaces to these.

If you want to work on the code, either for yourself or to contribute back to the project, clone the repository to your local machine with:

```
git clone https://github.com/datreant/datreant.git
```

CHAPTER 2

Contributing

This project is still under heavy development, and there are certainly rough edges and bugs. Issues and pull requests welcome! Check out our contributor's guide to learn how to get started with contributing back.

2.1 Installing datreant.data

Since `datreant.data` uses HDF5 as the file format of choice for persistence, you will first need to install the HDF5 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
```

You can then install `datreant.data` from PyPI using pip:

```
pip install datreant.data
```

It is also possible to use `--user` to install into your user's site-packages directory:

```
pip install --user datreant.data
```

2.1.1 Dependencies

The dependencies of `datreant.data` are:

- pandas: 0.16.1 or higher

- PyTables: 3.2.0 or higher
- h5py: 2.5.0 or higher

These are installed automatically when installing with pip.

2.1.2 Installing from source

To install from source, clone the repository and switch to the master branch

```
git clone git@github.com:datreant/datreant.data.git
cd datreant.data
git checkout master
```

Installation of the packages is as simple as

```
pip install .
```

This installs `datreant.data` in the system wide python directory; this may require administrative privileges.

It is also possible to use `--user` to install into your user's site-packages directory:

```
pip install --user .
```

2.2 Storing and retrieving datasets within Treants

The functionality of a `Treant` can be expanded to conveniently store `numpy` and `pandas` objects in a couple different ways. If we have an existing Treant:

```
>>> import datreant.core as dtr
>>> s = dtr.Treant('sequoia')
>>> s
<Treant: 'sequoia'>
```

We can attach the *`Data`* limb to only this instance with:

```
>>> import datreant.data
>>> s.attach('data')
>>> s.data
<Data([])>
```

Alternatively, we could attach the *`Data`* and *`AggData`* limbs to every object they apply for by doing:

```
>>> import datreant.data.attach
```

If you want explicit control of which objects have this limb, the first approach is the one to use, but the second one is useful for interactive work.

2.2.1 Storing and retrieving `numpy` arrays

Perhaps we have generated a numpy array of dimension (10^6, 3) that we wish to have easy access to later

```
>>> import numpy as np
>>> a = np.random.randn(1000000, 3)
>>> a.shape
(1000000, 3)
```

We can store this easily

```
>>> s.data['something wicked'] = a
>>> s.data
<Data(['something wicked'])>
```

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

```
>>> s.draw()
sequoia/
 +-- something wicked/
 |   +-- npData.h5
 +-- Treant.608f7463-5063-450a-96eb-c5c93f16dc32.json
```

and inside of this is a new HDF5 file (`npData.h5`). Our `numpy` array is stored inside, and we can recall it just as easily as we stored it:

```
>>> s.data['something wicked']
array([[ 0.49884872, -0.30062622,  0.64513512],
       [-0.12839311,  0.68467086, -0.96125085],
       [ 0.36655902, -0.13178154, -0.58137863],
       ...,
       [-0.20229488, -0.30303892,  1.44345568],
       [ 0.10119334, -0.50691484,  0.05854653],
       [-2.0551924 ,  0.80378532, -0.28869459]])
```

2.2.2 Storing and retrieving `pandas` objects

pandas is the *de facto* standard for working with tabular data in Python. It's most-used objects, the `Series` and `DataFrame` are just as easily stored as `numpy` arrays. If we have a DataFrame we wish to store:

```
>>> import pandas as pd
>>> df = pd.DataFrame(np.random.randn(1000, 3), columns=['A', 'B', 'C'])
>>> df.head()
          A         B         C
0 -0.474337 -1.257253  0.497824
1 -1.057806 -1.393081  0.628394
2  0.063369 -1.820173 -1.178128
3 -0.747949  0.607452 -1.509302
4 -0.031547 -0.680997  1.127573
```

then as you can expect, we can store it with:

```
>>> s.data['something terrible'] = df
```

and recall it with:

```
>>> s.data['something terrible'].head()
          A         B         C
```

(continues on next page)

(continued from previous page)

```
0 -0.474337 -1.257253  0.497824
1 -1.057806 -1.393081  0.628394
2  0.063369 -1.820173 -1.178128
3 -0.747949  0.607452 -1.509302
4 -0.031547 -0.680997  1.127573
```

Our data is stored in its own HDF5 file (`pdData.h5`) in the subdirectory we specified, so now our Treant looks like this:

```
s.draw()
sequoia/
 +-- something wicked/
 |   +-- npData.h5
 +-- Treant.608f7463-5063-450a-96eb-c5c93f16dc32.json
 +-- something terrible/
     +-- pdData.h5
```

Alternatively, we can use the *add()* method to store datasets:

```
>>> s.data.add('something terrible')
```

but the effect is the same. Since internally this uses the pandas.HDFStore class for storing pandas objects, all limitations for the types of indexes and objects it can store apply.

Appending to existing data

Sometimes we may have code that will generate a `Series` or `DataFrame` that is rather large, perhaps larger than our machine's memory. In these cases we can *append()* to an existing store instead of writing out a single, huge DataFrame all at once:

```
>>> s.data['something terrible'].shape      # before
(1000, 3)

>>> df2 = pd.DataFrame(np.random.randn(2000, 3), columns=['A', 'B', 'C'])
>>> s.data.append('something terrible', df2)
>>> s.data['something terrible'].shape      # after
(3000, 3)
```

Have code that will generate a DataFrame with 10^8 rows? No problem:

```
>>> for i in range(10**2):
...     a_piece = pd.DataFrame(np.random.randn(10**6, 3),
...                            columns=['A', 'B', 'C'],
...                            index=pd.Int64Index(np.arange(10**6) + i*10**6))
...
...     s.data.append('something enormous', a_piece)
```

Note that the `DataFrame` appended must have the same column names and dtypes as that already stored, and that only rows can be appended, not columns. For `pandas.Series` objects the dtype must match. Appending of `pandas.Panel` objects also works, but the limitations are more stringent. See the pandas HDFStore documentation for more details on what is technically possible.

Retrieving subselections

For pandas stores that are very large, we may not want or be able to pull the full object into memory. For these cases we can use *retrieve()* to get subselections of our data. Taking our large 10^8 row DataFrame, we can get at rows 1000000 to 2000000 with something like:

```
>>> s.data.retrieve('something enormous', start=10000000, stop=2000000).shape
(1000000, 3)
```

If we only wanted columns 'B' and 'C', we could get only those, too:

```
>>> s.data.retrieve('something enormous', start=10000000, stop=2000000,
...                 columns=['B', 'C']).shape
(1000000, 2)
```

These operations are performed "out-of-core", meaning that the full dataset is never read entirely into memory to get back the result of our subselection.

Retrieving from a query

For large datasets it can also be useful to retrieve only rows that match some set of conditions. We can do this with the `where` keyword, for example getting all rows for which column 'A' is less than -2:

```
>>> s.data.retrieve('something enormous', where="A < -2").head()
                 A         B         C
131      -2.177729 -0.797003  0.401288
134      -2.017321  0.750593 -1.366106
198      -2.203170 -0.670188  0.494191
246      -2.156695  1.107288 -0.065875
309      -2.334792  0.984636  0.006232
321      -3.784861 -1.222399  0.038717
346      -2.057103 -0.230953  0.732774
364      -2.418875  0.250880 -0.850418
413      -2.528563 -0.261624  1.233367
480      -2.205484  0.036570  0.501868
```

Note: Since our data is randomly generated in this example, the rows you get running the same example will be different.

Or perhaps when both column 'A' is less than -2 and column 'C' is greater than 2:

```
>>> s.data.retrieve('something enormous', where="A < -2 & C > 2").head()
                 A         B         C
1790     -3.103821 -0.616780  2.714530
5635     -2.431589 -0.580400  3.163408
7664     -2.364559  0.304764  2.884965
9208     -2.569256  1.105211  2.008396
9487     -2.028096  0.146484  2.234081
9968     -2.362063  0.544276  2.469602
11503    -2.494900 -0.005465  2.487311
12725    -2.353478 -0.001569  2.274861
14991    -2.129492 -1.889708  2.324640
15178    -2.327528  1.852786  2.425977
```

See the documentation for querying with pandas.HDFStore.select() for more information on the range of possibilities for the where keyword.

2.2.3 Bonus: storing anything pickleable

As a bit of a bonus, we can use the same basic storage and retrieval mechanisms that work for numpy and pandas objects to store Python object that is pickleable. For example, doing:

```
>>> s.data['a grocery list'] = ['ham', 'eggs', 'spam']
```

will store this list as a pickle:

```
>>> s.draw()
sequoia/
 +-- a grocery list/
 |   +-- pyData.pkl
 +-- something wicked/
 |   +-- npData.h5
 +-- Treant.608f7463-5063-450a-96eb-c5c93f16dc32.json
 +-- something enormous/
 |   +-- pdData.h5
 +-- something terrible/
     +-- pdData.h5
```

And we can get it back:

```
>>> s.data['a grocery list']
['ham', 'eggs', 'spam']
```

In this way we don't have to care too much about what type of object we are trying to store; the *Data* limb will try to pickle anything that isn't a numpy or pandas object.

2.2.4 Deleting datasets

We can delete stored datasets with the *remove()* method:

```
>>> s.data.remove('something terrible')
>>> s.draw()
sequoia/
 +-- a grocery list/
 |   +-- pyData.pkl
 +-- Treant.608f7463-5063-450a-96eb-c5c93f16dc32.json
 +-- something enormous/
 |   +-- pdData.h5
 +-- something wicked/
     +-- npData.h5
```

This will remove not only the file in which the data is actually stored, but also the directory if there are no other files present inside of it. If there are other files present, the data file will be deleted but the directory will not.

But since datasets live in the filesystem, we can also delete datasets by deleting them more directly, e.g. through a shell:

```
> rm -r sequoia/"something terrible"
```

and it will work just as well.

2.2.5 API reference: Data

See the *Data* API reference for more details.

2.3 Using Trees to subselect datasets

The `Data` limb isn't just for Treants; it works for `Tree` objects as well. So we could use our Treant 'sequoia' directly as a Tree instead of a Treant if we wanted:

```
>>> import datreant.core as dtr
>>> import datreant.data
>>> t = dtr.Tree('sequoia/')
>>> t.attach('data')
>>> t.data
<Data(['a grocery list', 'something enormous', 'something wicked'])>
```

and it would work all the same. This behavior is most useful, however, when nesting datasets.

2.3.1 Nesting within a tree

Dataset names are their paths downward relative to the Tree/Treant they are called from, so we can store a dataset like:

```
>>> t.data['a/better/grocery/list'] = ['ham', 'eggs', 'steak']
>>> t.data
<Data(['a grocery list', 'a/better/grocery/list', 'something enormous', 'something␣
↪wicked'])>
```

and this creates the directory structure you might expect:

```
>>> t.draw()
sequoia/
 +-- a grocery list/
 |   +-- pyData.pkl
 +-- Treant.608f7463-5063-450a-96eb-c5c93f16dc32.json
 +-- something enormous/
 |   +-- pdData.h5
 +-- a/
 |   +-- better/
 |       +-- grocery/
 |           +-- list/
 |               +-- pyData.pkl
 +-- something wicked/
     +-- npData.h5
```

This allows us to group together related datasets in a natural way, as we would probably do even if we weren't using `datreant` objects. So if we had several shopping lists, we might put them under a directory of their own:

```
>>> t.data['shopping lists/food'] = ['milk', 'ham', 'eggs', 'steak']
>>> t.data['shopping lists/clothes'] = ['shirts', 'pants', 'shoes']
>>> t.data['shopping lists/misc'] = ['dish soap']
```

which would give us:

```
>>> t['shopping lists'].draw()
shopping lists/
 +-- misc/
 |   +-- pyData.pkl
 +-- food/
 |   +-- pyData.pkl
 +-- clothes/
     +-- pyData.pkl
```

and we could always get them back easily enough:

```
>>> t.data['shopping lists/food']
['milk', 'ham', 'eggs', 'steak']
```

2.3.2 Trees as subselections

But since Trees can access datasets inside them, we could work more directly with our shopping lists by using the 'shopping lists' Tree

```
>>> lets_go_shopping = t['shopping lists']
>>> lets_go_shopping.data
 <Data(['clothes', 'food', 'misc'])>
```

and now selecting is a bit less verbose:

```
>>> lets_go_shopping['food']
['milk', 'ham', 'eggs', 'steak']
```

2.4 Aggregating datasets with Views and Bundles

Just as Treants and Trees have the *Data* limb for storing and retrieving datasets in their filesystem trees, the `View` and `Bundle` objects also have the *AggData* limb for accessing these datasets in aggregate.

Given a directory with four Treants

```
> ls
elm/  maple/  oak/  sequoia/
```

we'll gather these up in a Bundle

```
>>> import datreant.core as dtr
>>> import glob
>>> b = dtr.Bundle(glob.glob('*'))
>>> b
<Bundle([<Treant: 'sequoia'>, <Treant: 'maple'>, <Treant: 'oak'>, <Treant: 'elm'>])>
```

and then attach the `AggData` limb to only this Bundle instance with:

```
>>> import datreant.data
>>> b.attach('data')
```

Note: Attaching a limb like *AggData* to a Bundle or View with the `attach()` method will attach the required limb to each member instance. In this case, each member gets a *Data* limb.

and so we can now do:

```
>>> b.data
<AggData([])>
```

This tells us that there are no datasets with the same key within every member of the Bundle. So, let's make something that does. Let's build a "dataset" that gives us a sinusoid based on a characteristic of each Treant in the Bundle:

```
>>> import numpy as np
>>> b.categories['frequency'] = [1, 2, 3, 4]
>>> for member in b:
...     member.data['sinusoid/array'] = np.sin(
...         member.categories['frequency'] * np.linspace(0, 8*np.pi,
...                                                      num=200))
```

So now if we do:

```
>>> b.data
<AggData(['sinusoid/array'])>
```

we see we now have a dataset name in common among all members. If we recall it

```
>>> sines = b.data['sinusoid/array']
>>> type(sines)
dict
```

we get back a dictionary with the full path to each member as keys:

```
>>> sines.keys()
['/home/bob/research/arborea/sequoia/',
 '/home/bob/research/arborea/oak/',
 '/home/bob/research/arborea/elm/',
 '/home/bob/research/arborea/maple/']
```

and the values are the :*numpy* arrays we stored for each member. If we'd rather get back a dictionary with names instead of paths, we could do that with the *retrieve()* method:

```
>>> b.data.retrieve('sinusoid/array', by='name').keys()
['sequoia', 'oak', 'maple', 'elm']
```

Getting uuids as the keys is also possible, and is often useful since these will be unique among Treants, while names (and in some cases, paths) are generally not.

2.4.1 Aggregating datasets not represented among all members

We can still aggregate over datasets even if their keys are not present among all members. We can see what keys are available among at least one member in the Bundle with:

```
>>> b.data.any
['a grocery list',
'a/better/grocery/list',
```

(continues on next page)

(continued from previous page)

```
'shopping lists/clothes',
'shopping lists/food',
'shopping lists/misc',
'sinusoid/array',
'something enormous',
'something wicked']
```

and we see the datasets we stored using the single Treant earlier. If we recall one of these, we get an aggregation

```
>>> b.data['shopping lists/clothes']
{'/home/bob/research/arborea/sequoia/': ['shirts', 'pants', 'shoes']}
```

with only the datasets present for that key. Since it's only the one Treant that has a dataset with this name, we get a dictionary with one key-value pair.

2.4.2 MultiIndex aggregation for `pandas` objects

`numpy` arrays or pickled datasets are always retrieved in aggregate as dictionaries, since this is the simplest way of aggregating these objects while retaining the ability to identify datasets from individual members. Aggregation is most useful, however, for `pandas` objects, since for these we can naturally build versions of the same data structure with an additional index for data membership.

We'll make a `pandas.Series` version of the same dataset we stored before:

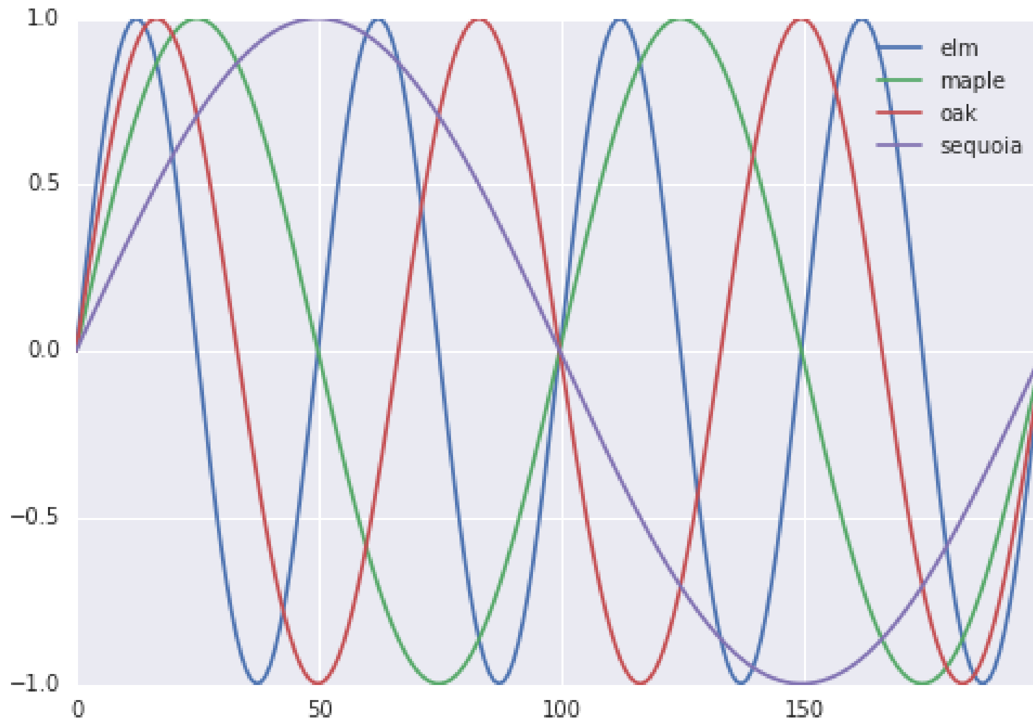
```
>>> import pandas as pd
>>> for member in b:
...     member.data['sinusoid/series'] = pd.Series(member.data['sinusoid/array'])
```

So now when we retrieve this aggregated dataset by name, we get a series with an outermost index of member names:

```
>>> sines = b.data.retrieve('sinusoid/series', by='name')
>>> sines.groupby(level=0).head()
sequoia  0    0.000000
         1    0.125960
         2    0.249913
         3    0.369885
         4    0.483966
oak      0    0.000000
         1    0.369885
         2    0.687304
         3    0.907232
         4    0.998474
maple    0    0.000000
         1    0.249913
         2    0.483966
         3    0.687304
         4    0.847024
elm      0    0.000000
         1    0.483966
         2    0.847024
         3    0.998474
         4    0.900479
dtype: float64
```

So we can immediately use this for aggregated analysis, or perhaps just pretty plots:

```
>>> for name, group in sines.groupby(level=0):
...     group.reset_index(level=0, drop=True).plot(legend=True, label=name)
```



2.4.3 Subselection with Views

Just as we can *subselect datasets with Trees*, we can use `View` objects to work with subselections in aggregate. Using our Bundle from above, we can construct a View:

```
>>> sinusoids = dtr.View(b).trees['sinusoid']
>>> sinusoids
<View([<Tree: 'sequoia/sinusoid/'>, <Tree: 'maple/sinusoid/'>, <Tree: 'oak/sinusoid/'>
↪, <Tree: 'elm/sinusoid/'>])>
```

And just like a `Tree` can access datasets with the *`Data`* limb in the same way a `Treant` can, a `View` can access datasets in aggregate in the same way as a `Bundle`:

```
>>> sinusoids.attach('data')
>>> sinusoids.data
<AggData(['array', 'series'])>
```

These are the datasets common to all the Trees in this View. We can retrieve an aggregation as before:

```
>>> sinusoids.data['series'].groupby(level=0).head()
/home/bob/research/arborea/sequoia/sinusoid/  0    0.000000
                                              1    0.031569
```

(continues on next page)

(continued from previous page)

```
                                               2    0.063106
                                               3    0.094580
                                               4    0.125960
/home/bob/research/arborea/maple/sinusoid/     0    0.000000
                                               1    0.063106
                                               2    0.125960
                                               3    0.188312
                                               4    0.249913
/home/bob/research/arborea/oak/sinusoid/       0    0.000000
                                               1    0.094580
                                               2    0.188312
                                               3    0.280355
                                               4    0.369885
/home/bob/research/arborea/elm/sinusoid/       0    0.000000
                                               1    0.125960
                                               2    0.249913
                                               3    0.369885
                                               4    0.483966
dtype: float64
```

Note: For aggregations from a View, it is not possible to aggregate by uuid because Trees do not have them. Also, in many cases, as here, aggregating by name will not give unique keys. When the aggregation keys are not unique, a `KeyError` is raised.

2.4.4 API reference: AggData

See the *AggData* API reference for more details.

2.5 API Reference

This is an overview of the `datreant.data` API components.

2.5.1 Individual datasets

These are the API components of `datreant.data` for storing and retrieving datasets from individual Treants and Trees.

Data

The class `datreant.data.limbs.Data` is the interface used by Treants to access their stored datasets.

class datreant.data.limbs.**Data**(*tree*)
Interface to stored data.

add(*handle*, **args*, ***kwargs*)
Store data in Treant.

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

keys()

List available datasets.

Returns

handles list of handles to available datasets

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 `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

2.5.2 Aggregated data

These are the API components of `datreant.data` for working with datasets from multiple Treants at once, and treating them in aggregate.

AggData

The class *`datreant.data.agglimbs.AggData`* is the interface used by Bundles and Views to access their members' datasets in aggregate.

class `datreant.data.agglimbs.`**`AggData`**(*collection*)

Manipulators for collection data.

`keys`(*scope='all'*)

List available datasets.

Parameters **`scope`** (*{'all', 'any'}*) – Keys to list. 'all' returns only handles that are present in all members. 'any' returns a list of all handles present in at least one member.

Returns **handles** – list of handles to available datasets

Return type list

`retrieve`(*handle*, *by='path'*, ***kwargs*)

Retrieve aggregated dataset from all members.

This is a convenience method. The stored data structure for each member is read from disk and aggregated. The aggregation scheme is dependent on the form of the data structures pulled from each member:

pandas DataFrames or Series the structures are appended together, with a new level added to the index giving the member (see *by*) each set of rows came from

pandas Panel or Panel4D, numpy arrays, pickled python objects the structures are returned as a dictionary, with keys giving the member (see *by*) and each value giving the corresponding data structure

This method tries to do smart things with the data it reads from each member. In particular:

- members for which there is no data with the given handle are skipped
- the lowest-common-denominator data structure is output; this means that if all data structures read are pandas DataFrames, then a multi-index DataFrame is returned; if some structures are pandas DataFrames, while some are anything else, a dictionary is returned

Arguments

handle name of data to retrieve

Keywords

by top-level index of output data structure; ‘path’ uses member path, ‘name’ uses member names, ‘uuid’ uses member uuids; if names are not unique, it is better to go with ‘path’ or ‘uuid’ [‘path’]

See `datreant.data.limbs.Data.retrieve()` for more information on keyword usage.

Keywords for pandas data structures

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 aggregated data structure

Index

A

add() (datreant.data.limbs.Data method), 16
AggData (class in datreant.data.agglimbs), 18
append() (datreant.data.limbs.Data method), 17

D

Data (class in datreant.data.limbs), 16

K

keys() (datreant.data.agglimbs.AggData method), 18
keys() (datreant.data.limbs.Data method), 17

R

remove() (datreant.data.limbs.Data method), 17
retrieve() (datreant.data.agglimbs.AggData method), 18
retrieve() (datreant.data.limbs.Data method), 17