datreant Documentation

Release 0.6.0

David Dotson

March 25, 2016

In many fields of science, especially those analyzing experimental or simulation data, there is often an existing ecosystem of specialized tools and file formats which new tools must work around, for better or worse. Furthermore, centralized database solutions may be suboptimal for data storage for a number of reasons, including insufficient hardware infrastructure, variety and heterogeneity of raw data, the need for data portability, etc. This is particularly the case for fields centered around simulation: simulation systems can vary widely in size, composition, rules, paramaters, and starting conditions. And with increases in computational power, it is often necessary to store intermediate results obtained from large amounts of simulation data so it can be accessed and explored interactively.

These problems make data management difficult, and serve as a barrier to answering scientific questions. To make things easier, `datreant` is a collection of Python packages that provide a pythonic interface to the filesystem and the data that lives within it. It solves a boring problem, so we can focus on interesting ones.

CHAPTER 1

Stay organized

`datreant` offers a layer of flexibility and sanity to the task of analyzing data from many studies, whether they be individual simulations or data from field work. Its core object, the **Treant**, is designed to be subclassed: the classes in datreant are useful on their own but vanilla by design, and are built to be easily extended into domain-specific objects.

As an example: MDSynthesis, a package for storing, recalling, and aggregating data from molecular dynamics simulations, is built on top of datreant.

CHAPTER 2

The datreant namespace

`datreant` is a namespace package, which means that it's more a package of packages. These packages are all dependent on a central, core library, called `datreant.core`. This documentation is for that core library.

Other packages in the `datreant` namespace currently include:

- datreant.data

CHAPTER 3

Getting datreant

See the installation instructions for installation details. The package itself is pure Python, and light on dependencies by design.

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.core.git
```

CHAPTER 4

Contributing

This project is still under heavy development, and there are certainly rough edges and bugs. Issues and pull requests welcome! See *Contributing to datreant* for more information.

4.1 Installing datreant.core

You can install `datreant.core` from PyPI using pip:

```
pip install datreant.core
```

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

```
pip install --user datreant.core
```

All datreant packages currently support the following Python versions:

```
- 2.7
- 3.3
- 3.4
- 3.5
```

4.1.1 Dependencies

The dependencies of `datreant.core` are light, with many being pure-Python packages themselves. The current dependencies are:

```
- asciitree
- pathlib
- scandir
- six
- fuzzywuzzy
```

These are automatically installed when installing `datreant.core`.

4.1.2 Installing from source

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

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

Installation of the packages is as simple as

```
pip install .
```

This installs datreant in the system wide python directory; this may require administrative privileges. If you have a virtualenv active, it will install the package within your virtualenv. See *Setting up your development environment* for more on setting up a proper development environment.

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

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

4.2 Creating and using Treants

datreant is not an analysis library. 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.

The basic functionality of datreant is condensed into one object: the **Treant**. Named after the talking trees of D&D lore, Treants are persistent objects that live as directory trees in the filesystem and store their state information 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 Treant at the same time.

> **Warning:** File locking is performed with POSIX advisory locks. These are not guaranteed to work perfectly on all platforms and file systems, so use caution when changing the stored attributes of a Treant in more than one process. Also, though advisory locks are mostly process safe, they are definitely not thread safe. Don't use multithreading and try to modify Treant elements at the same time.

4.2.1 Persistence as a feature

Treants store their data as directory structures in the file system. Generating a new Treant, for example, with the following

```
>>> # python session 1
>>> import datreant.core as dtr
>>> s = dtr.Treant('sprout')
```

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

```
> # shell
> ls sprout
Treant.2b4b5800-48a7-4814-ba6d-1e631a09a199.json
```

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

```
>>> # python session 2
>>> import datreant.core as dtr
>>> s = dtr.Treant('sprout')
```

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

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

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

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 Treant in this example will certainly differ from any Treants you generate. This is used to differentiate Treants from each other. Unexpected and broken behavior will result from changing the names of state files!

4.2.2 What goes into a state file?

The state file of a Treant contains the core pieces of information that define it. A few of these things are defined in the filesystem itself, including

```
/home/bob/research/arborea/sprout/Treant.2b4b5800-48a7-4814-ba6d-1e631a09a199.json
|_____________________________|_______|_______|_____________________________________|
          location            name      ^                  uuid
|_____________________________________|    |
             abspath                 treanttype
|_____________________________________________________________________________________|
                                     filepath
```

This means that changing the location or name of a Treant can be done at the filesystem level. Although this means that one can change the treanttype and uuid as well, this is generally not recommended.

Other components, such as the Treant's tags and categories, are stored internally in the state file (see *Differentiating Treants* for more on these).

4.2.3 API Reference: Treant

See the *Treant* API reference for more details.

4.3 Differentiating Treants

Treants can be used to develop "fire-and-forget" analysis routines. Large numbers of Treants can be fed to an analysis routine, with individual Treants handled according to their characteristics. To make it possible to write code that tailors its approach according to the Treant it encounters, we can use tags and categories.

4.3.1 Using tags

Tags are individual strings that describe a Treant. Using our Treant `sprout` as an example, we can add many tags at once

```
>>> from datreant import Treant
>>> s = Treant('sprout')
>>> s.tags.add('elm', 'mirky', 'misty')
>>> s.tags
<Tags(['elm', 'mirky', 'misty'])>
```

They can be iterated through as well

```
>>> for tag in s.tags:
>>>     print tag
elm
mirky
misty
```

Or checked for membership

```
>>> 'mirky' in s.tags
True
```

Tags function as sets

Since the tags of a Treant behave as a set, we can do set operations directly, such as subset comparisons:

```
>>> {'elm', 'misty'} < s.tags
True
```

unions:

```
>>> {'moldy', 'misty'} | s.tags
{'elm', 'mirky', 'misty', 'moldy'}
```

intersections:

```
>>> {'elm', 'moldy'} & s.tags
{'elm'}
```

differences:

```
>>> s.tags - {'moldy', 'misty'}
{'elm', 'mirky'}
```

or symmetric differences:

```
>>> s.tags ^ {'moldy', 'misty'}
{u'elm', u'mirky', 'moldy'}
```

It is also possible to set the tags directly:

```
>>> s.tags = s.tags | {'moldy', 'misty'}
>>> s.tags
<Tags(['elm', 'mirky', 'misty', 'moldy'])>
```

API Reference: Tags

See the *Tags* API reference for more details.

4.3.2 Using categories

Categories are key-value pairs. They are particularly useful as switches for analysis code. For example, if we have Treants with different shades of bark (say, "dark" and "light"), we can make a category that reflects this. In this case, we categorize `sprout` as "dark"

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

Perhaps we've written some analysis code that will take both "dark" and "light" Treants as input but needs to handle them differently. It can see what variety of **Treant** it is working with using

```
>>> s.categories['bark']
'dark'
```

The keys for categories must be strings, but the values may be strings, numbers (floats, ints), or booleans (`True`, `False`).

Note: `None` may not be used as a category value since this is used in aggregations (see *Coordinating Treants with Bundles*) to indicate keys that are absent.

API Reference: Categories

See the *Categories* API reference for more details.

4.3.3 Filtering and grouping on tags and categories

Tags and categories are especially useful for filtering and grouping Treants. See *Coordinating Treants with Bundles* for the details on how to flexibly do this.

4.4 Filesystem manipulation with Trees and Leaves

A Treant functions as a specially marked directory, having a state file with identifying information. What's a Treant without a state file? It's just a **Tree**.

datreant gives pythonic access to the filesystem by way of **Trees** and **Leaves** (directories and files, respectively). Say our current working directory has two directories and a file

```
> ls
moe/   larry/   curly.txt
```

We can use Trees and Leaves directly to manipulate them

```
>>> import datreant.core as dtr
>>> t = dtr.Tree('moe')
>>> t
<Tree: 'moe'>

>>> l = dtr.Leaf('curly.txt')
>>> l
<Leaf: 'curly.txt'>
```

These objects point to a specific path in the filesystem, which doesn't necessarily have to exist. Just as with Treants, more than one instance of a Tree or Leaf can point to the same place.

4.4.1 Working with Trees

Tree objects can be used to introspect downward into their directory structure. Since a Tree is essentially a container for its own child Trees and Leaves, we can use getitem syntax to dig around

```
>>> t = dtr.Tree('moe')
>>> t['a/directory/']
<Tree: 'moe/a/directory/'>

>>> t['a/file']
<Leaf: 'moe/a/file'>
```

Paths that resolve as being inside a Tree give *True* for membership tests

```
>>> t['a/file'] in t
True
```

Note that these items need not exist

```
>>> t['a/file'].exists
False
```

in which case whether a Tree or Leaf is returned is dependent on an ending `/`. We can create directories and empty files easily enough, though:

```
>>> adir = t['a/directory/'].make()
>>> adir.exists
True

>>> afile = t['a/file'].make()
>>> afile.exists
True
```

Note: For accessing directories and files that exist, getitem syntax isn't sensitive to ending `/` separators to determine whether to give a Tree or a Leaf.

API Reference: Tree

See the *Tree* API reference for more details.

4.4.2 A Treant is a Tree

The **Treant** object is a subclass of a Tree, so the above all applies to Treant behavior. Some methods of Trees are especially useful when working with Treants. One of these is `draw`

```
>>> s = dtr.Treant('sprout')
>>> s['a/new/file'].make()
>>> s['a/.hidden/directory/'].make()
>>> s.draw()
sprout/
 +-- Treant.839c7265-5331-4224-a8b6-c365f18b9997.json
```

```
 +-- a/
     +-- new/
     |   +-- file
     +-- .hidden/
         +-- directory/
```

which gives a nice ASCII-fied visual of the Tree. We can also obtain a collection of Trees and/or Leaves in the Tree with globbing

```
>>> s.glob('a/*')
<View([<Tree: 'sprout/a/.hidden/'>, <Tree: 'sprout/a/new/'>])>
```

See *Using Views to work with Trees and Leaves collectively* for more about the **View** object, and how it can be used to manipulate many Trees and Leaves as a single logical unit. More details on how to introspect Trees with Views can be found in *Views from a Tree*.

4.4.3 File operations with Leaves

Leaf objects are interfaces to files. At the moment they are most useful as pointers to particular paths in the filesystem, making it easy to save things like plots or datasets within the Tree they need to go:

```
>>> import numpy as np
>>> random_array = np.random.randn(1000, 3)
>>> np.save(t['random/array.npy'].makedirs().abspath, random_array)
```

Or getting things back later:

```
>>> np.load(t['random/array.npy'].abspath)
array([[ 1.28609187, -0.08739047,  1.23335427],
       [ 1.85979027,  0.37250825,  0.89576077],
       [-0.77038908, -0.02746453, -0.13723022],
       ...,
       [-0.76445797,  0.94284523,  0.29052753],
       [-0.44437005, -0.91921603, -0.4978258 ],
       [-0.70563139, -0.62811205,  0.60291534]])
```

But they can also be used for introspection, such as reading the bytes from a file:

```
>>> t['about_moe.txt'].read()
'Moe is not a nice person.\n'
```

API Reference: Leaf

See the *Leaf* API reference for more details.

4.5 Using Views to work with Trees and Leaves collectively

A **View** makes it possible to work with arbitrary Trees and Leaves as a single logical unit. It is an ordered set of its members.

4.5.1 Building a View and selecting members

Views can be built from a list of paths, existing or not. Taking our working directory with

```
> ls
moe/   larry/   curly.txt
```

We can build a View immediately

```
>>> import datreant.core as dtr
>>> import glob
>>> v = dtr.View(glob.glob('*'))
>>> v
<View([<Tree: 'moe/'>, <Tree: 'larry/'>, <Leaf: 'curly.txt'>])>
```

And we can get to work using it. Since Views are firstly a collection of members, individual members can be accessed through indexing and slicing

```
>>> v[1]
<Tree: 'larry/'>

>>> v[1:]
<View([<Tree: 'larry/'>, <Leaf: 'curly.txt'>])>
```

But we can also use fancy indexing (which can be useful for re-ordering members)

```
>>> v[[2, 0]]
<View([<Leaf: 'curly.txt'>, <Tree: 'moe/'>])>
```

Or boolean indexing

```
>>> v[[True, False, True]]
<View([<Tree: 'moe/'>, <Leaf: 'curly.txt'>])>
```

As well as indexing by name

```
>>> v['curly.txt']
<View([<Leaf: 'curly.txt'>])>
```

Note that since the name of a file or directory need not be unique, this always returns a View.

4.5.2 Filtering View members

Often we might obtain a View of a set of files and directories and then use the View itself to filter down into the set of things we actually want. There are a number of convenient ways to do this.

Want only the Trees?

```
>>> v.membertrees
<View([<Tree: 'moe/'>, <Tree: 'larry/'>])>
```

Or only the Leaves?

```
>>> v.memberleaves
<View([<Leaf: 'curly.txt'>])>
```

We can get more granular and filter members using glob patterns on their names:

```
>>> v.filter('*r*')
<View([<Tree: 'larry/'>, <Leaf: 'curly.txt'>])>
```

And since all these properties and methods return Views, we can stack operations:

```
>>> v.filter('*r*').memberleaves
<View([<Leaf: 'curly.txt'>])>
```

4.5.3 Views from a Tree

A common use of a View is to introspect the children of a Tree. If we have a look inside one of our directories

```
> ls moe/
about_moe.txt  more_moe.pdf  sprout/
```

We find two files and a directory. We can get at the files with

```
>>> moe = v['moe'][0]
>>> moe.leaves
<View([<Leaf: 'moe/about_moe.txt'>, <Leaf: 'moe/more_moe.pdf'>])>
```

and the directories with

```
>>> moe.trees
<View([<Tree: 'moe/sprout/'>])>
```

Both these properties leave out hidden children, since hidden files are often hidden to keep them out of the way of most work. But we can get at these easily, too:

```
>>> moe.hidden
<View([<Leaf: 'moe/.hiding_here'>])>
```

Want all the children?

```
>>> moe.children
<View([<Tree: 'moe/sprout/'>, <Leaf: 'moe/about_moe.txt'>,
       <Leaf: 'moe/more_moe.pdf'>, <Leaf: 'moe/.hiding_here'>])>
```

4.5.4 A View is an ordered set

Because a View is a set, adding members that are already present results in no change to the View:

```
>>> v.add('moe')
>>> v
<View([<Tree: 'moe/'>, <Tree: 'larry/'>, <Leaf: 'curly.txt'>])>

# addition of Trees/Leaves to a View also returns a View
>>> v + dtr.Tree('moe')
<View([<Tree: 'moe/'>, <Tree: 'larry/'>, <Leaf: 'curly.txt'>])>
```

But a View does have a sense of order, so we could, for example, meaningfully get a View with the order of members reversed:

```
>>> v[::-1]
<View([<Leaf: 'curly.txt'>, <Tree: 'larry/'>, <Tree: 'moe/'>])>
```

Because it is functionally a set, operations between Views work as expected. Making another View with

```
>>> v2 = dtr.View('moe', 'nonexistent_file.txt')
```

we can get the union:

```
>>> v | v2
<View([<Tree: 'moe/'>, <Tree: 'larry/'>, <Leaf: 'curly.txt'>,
       <Leaf: 'nonexistent_file.txt'>])>
```

the intersection:

```
>>> v & v2
<View([<Tree: 'moe/'>])>
```

differences:

```
>>> v - v2
<View([<Tree: 'larry/'>, <Leaf: 'curly.txt'>])>

>>> v2 - v
<View([<Leaf: 'nonexistent_file.txt'>])>
```

or the symmetric difference:

```
>>> v ^ v2
<View([<Leaf: 'curly.txt'>, <Tree: 'larry/'>,
       <Leaf: 'nonexistent_file.txt'>])>
```

4.5.5 Collective properties and methods of a View

A View is a collection of Trees and Leaves, but it has methods and properties that mirror those of Trees and Leaves that allow actions on all of its members in aggregate. For example, we can directly get all directories and files within each member Tree:

```
>>> v.children
<View([<Tree: 'moe/sprout/'>, <Leaf: 'moe/about_moe.txt'>,
       <Leaf: 'moe/more_moe.pdf'>, <Leaf: 'moe/.hiding_here'>,
       <Leaf: 'larry/about_larry.txt'>])>
```

Or we could get all children that match a glob pattern:

```
>>> v.glob('*moe*')
<View([<Leaf: 'moe/about_moe.txt'>, <Leaf: 'moe/more_moe.pdf'>])>
```

Note that this is the equivalent of doing something like:

```
>>> dtr.View([tree.glob(pattern) for tree in v.membertrees])
```

In this way, a View functions analogously for Trees and Leaves as a Bundle does for Treants. See *Coordinating Treants with Bundles* for more on this theme.

4.5.6 API Reference: View

See the *View* API reference for more details.

4.6 Coordinating Treants with Bundles

Similar to a View, a **Bundle** is an ordered set of Treants that makes it easy to work with them as a single logical unit. Bundles can be constructed in a variety of ways, but often with a collection of Treants. If our working directory has a few Treants in it:

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

We can make a Bundle with

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

Bundles can also be initialized from existing Treant instances, in addition to their paths in the filesystem, so

```
>>> t = dtr.Treant('elm')
>>> b = dtr.Bundle(t, 'maple', 'oak', 'sequoia')
```

would work equally well.

4.6.1 Gathering Treants from the filesystem

It can be tedious manually hunting for existing Treants throughout the filesystem. For this reason the *discover()* function can do this work for us:

```
>>> b = dtr.discover('.')
>>> b
<Bundle([<Treant: 'sequoia'>, <Treant: 'maple'>, <Treant: 'oak'>, <Treant: 'elm'>])>
```

For this simple example all our Treants were in this directory, so it's not quite as useful. But for a directory structure that is deep and convoluted perhaps from a project spanning years, *discover()* lets you get a Bundle of all Treants in the tree with little effort. You can then filter on tags and categories to get Bundles of the Treants you actually want to work with.

See the *datreant.core.discover()* API reference for more details.

4.6.2 Basic member selection

All the same selection patterns that work for Views (see *Building a View and selecting members*) work for Bundles. This includes indexing with integers:

```
>>> b = dtr.discover()
>>> b[1]
<Treant: 'maple'>
```

slicing:

```
>>> b[1:]
<Bundle([<Treant: 'maple'>, <Treant: 'oak'>, <Treant: 'elm'>])>
```

fancy indexing:

```
>>> b[[1, 2, 0]]
<Bundle([<Treant: 'maple'>, <Treant: 'oak'>, <Treant: 'sequoia'>])>
```

boolean indexing:

```
>>> b[[False, False, True, False]]
<Bundle([<Treant: 'oak'>])>
```

and indexing by Treant name:

```
>>> b['oak']
<Bundle([<Treant: 'oak'>])>
```

Note that since Treant names need not be unique, indexing by name always yields a Bundle.

4.6.3 Filtering on Treant tags

Treants are more useful than plain Trees because they carry distinguishing characteristics beyond just their path in the filesystem. Tags are one of these distinguishing features, and Bundles can use them directly to filter their members.

Note: For a refresher on using tags with individual Treants, see *Using tags*. Everything that applies to using tags with individual Treants applies to using them in aggregate with Bundles.

The aggregated tags for all members in a Bundle are accessible via *datreant.core.Bundle.tags*. Just calling this property gives a view of the tags present in every member Treant:

```
>>> b.tags
<AggTags(['plant'])>
```

But our Treants probably have more than just this one tag. We can get at the tags represented by at least one Treant in the Bundle with

```
>>> b.tags.any
{'building',
 'firewood',
 'for building',
 'furniture',
 'huge',
 'paper',
 'plant',
 'shady',
 'syrup'}
```

Since tags function as a set, we get back a set. Likewise we have

```
>>> b.tags.all
{'plant'}
```

which we've already seen.

Using tag expressions to select members

We can use getitem syntax to query the members of Bundle. For example, giving a single tag like

```
>>> b.tags['building']
[False, False, True, True]
```

gives us back a list of booleans. This can be used directly on the Bundle as a boolean index to get back a subselection of its members:

```
>>> b[b.tags['building']]
<Bundle([<Treant: 'oak'>, <Treant: 'elm'>])>
```

We can also provide multiple tags to match more Treants:

```
>>> b[b.tags['building', 'furniture']]
<Bundle([<Treant: 'maple'>, <Treant: 'oak'>, <Treant: 'elm'>])>
```

The above is equivalent to giving a tuple of tags to match, as below:

```
>>> b[b.tags[('building', 'furniture')]]
<Bundle([<Treant: 'maple'>, <Treant: 'oak'>, <Treant: 'elm'>])>
```

Using a tuple functions as an "or"-ing of the tags given, in which case the resulting members are those that have at least one of the tags inside the tuple.

But if we give a list instead, we get:

```
>>> b[b.tags[['building', 'furniture']]]
<Bundle([])>
```

...something else, in this case nothing. Giving a list functions as an "and"-ing of the tags given inside, so the above query will only give members that have both 'building' and 'furniture' as tags. There were none in this case.

Lists and tuples can be nested to build complex and/or selections. In addition, sets can be used to indicate negation ("not"):

```
>>> b[b.tags[{'furniture'}]]
<Bundle([<Treant: 'sequoia'>, <Treant: 'oak'>, <Treant: 'elm'>])>
```

Putting multiple tags inside a set functions as a negated "and"-ing of the contents:

```
>>> b[b.tags[{'building', 'furniture'}]]
<Bundle([<Treant: 'sequoia'>, <Treant: 'maple'>, <Treant: 'oak'>, <Treant: 'elm'>])>
```

which is the opposite of the empty Bundle we got when we did the "and"-ing of these tags earlier.

Fuzzy matching for tags

Over the course of a project spanning years, you might add several variations of essentially the same tag to different Treants. For example, it looks like we might have two different tags that mean the same thing among the Treants in our Bundle:

```
>>> b.tags
{'building',
 'firewood',
 'for building',
 'furniture',
 'huge',
 'paper',
 'plant',
 'shady',
 'syrup'}
```

Chances are good we meant the same thing when we added 'building' and 'for building' to these Treants. How can we filter on these without explicitly including each one in a tag expression?

We can use fuzzy matching:

```
>>> b.tags.fuzzy('building', scope='any')
('building', 'for building')
```

which we can use directly as an "or"-ing in a tag expression:

```
>>> b[b.tags[b.tags.fuzzy('building', scope='any')]]
<Bundle([<Treant: 'oak'>, <Treant: 'elm'>])>
```

The threshold for fuzzy matching can be set with the `threshold` parameter. See the API reference for *`fuzzy()`* for more details on how to use this method.

4.6.4 Grouping with Treant categories

Besides tags, categories are another mechanism for distinguishing Treants from each other. We can access these in aggregate with a Bundle, but we can also use them to build groupings of members by category value.

Note: For a refresher on using categories with individual Treants, see *Using categories*. Much of what applies to using categories with individual Treants applies to using them in aggregate with Bundles.

The aggregated categories for all members in a Bundle are accessible via *`datreant.core.Bundle.categories`*. Just calling this property gives a view of the categories with keys present in every member Treant:

```
>>> b.categories
<AggCategories({'age': ['adult', 'young', 'young', 'old'],
                'type': ['evergreen', 'deciduous', 'deciduous', 'deciduous'],
                'bark': ['fibrous', 'smooth', 'mossy', 'mossy']})>
```

We see that here, the values are lists, with each element of the list giving the value for each member, in member order. This is how categories behave when accessing from Bundles, since each member may have a different value for a given key.

But just as with tags, our Treants probably have more than just the keys 'age', 'type', and 'bark' among their categories. We can get a dictionary of the categories with each key present among at least one member with

```
>>> b.categories.any
{'age': ['adult', 'young', 'young', 'old'],
 'bark': ['fibrous', 'smooth', 'mossy', 'mossy'],
 'health': [None, None, 'good', 'poor'],
 'nickname': ['redwood', None, None, None],
 'type': ['evergreen', 'deciduous', 'deciduous', 'deciduous']}
```

Note that for members that lack a given key, the value returned in the corresponding list is `None`. Since `None` is not a valid value for a category, this unambiguously marks the key as being absent for these members.

Likewise we have

```
>>> b.categories.all
{'age': ['adult', 'young', 'young', 'old'],
 'bark': ['fibrous', 'smooth', 'mossy', 'mossy'],
 'type': ['evergreen', 'deciduous', 'deciduous', 'deciduous']}
```

which we've already seen.

Accessing and setting values with keys

Consistent with the behavior shown above, when accessing category values in aggregate with keys, what is returned is a list of values for each member, in member order:

```
>>> b.categories['age']
['adult', 'young', 'young', 'old']
```

And if we access a category with a key that isn't present among all members, `None` is given for those members in which it's missing:

```
>>> b.categories['health']
[None, None, 'good', 'poor']
```

If we're interested in the values corresponding to a number of keys, we can access these all at once with either a list:

```
>>> b.categories[['health', 'bark']]
[[None, None, 'good', 'poor'], ['fibrous', 'smooth', 'mossy', 'mossy']]
```

which will give a list with the values for each given key, in order by key. Or with a set:

```
>>> b.categories[{'health', 'bark'}]
{'bark': ['fibrous', 'smooth', 'mossy', 'mossy'],
 'health': [None, None, 'good', 'poor']}
```

which will give a dictionary, with keys as keys and values as values.

We can also set category values for all members as if we were working with a single member:

```
>>> b.categories['height'] = 'tall'
>>> b.categories['height']
['tall', 'tall', 'tall', 'tall']
```

or we could set the value for each member:

```
>>> b.categories['height'] = ['really tall', 'middling', 'meh', 'tall']
>>> b.categories['height']
['really tall', 'middling', 'meh', 'tall']
```

Grouping by value

Since for a given key a Bundle may have members with a variety of values, it can be useful to get subsets of the Bundle as a function of value for a given key. We can do this using the *groupby()* method:

```
>>> b.categories.groupby('type')
{'deciduous': <Bundle([<Treant: 'maple'>, <Treant: 'oak'>, <Treant: 'elm'>])>,
 'evergreen': <Bundle([<Treant: 'sequoia'>])>}
```

In grouping by the 'type' key, we get back a dictionary with the values present for this key as keys and Bundles giving the corresponding members as values. We could iterate through this dictionary and apply different operations to each Bundle based on the value. Or we could extract out only the subset we want, perhaps just the 'deciduous' Treants:

```
>>> b.categories.groupby('type')['deciduous']
<Bundle([<Treant: 'maple'>, <Treant: 'oak'>, <Treant: 'elm'>])>
```

We can also group by more than one key at once:

```
>>> b.categories.groupby(['type', 'health'])
{('good', 'deciduous'): <Bundle([<Treant: 'oak'>])>,
 ('poor', 'deciduous'): <Bundle([<Treant: 'elm'>])>}
```

Now the keys of the resulting dictionary are tuples of value combinations for which there are members. The resulting Bundles don't include some members since not every member has both the keys 'type' and 'health'.

See the API reference for *groupby()* for more details on how to use this method.

4.6.5 Operating on members in parallel

Although it's common to iterate through the members of a Bundle to perform operations on them individually, this approach can often be put in terms of mapping a function to each member independently. A Bundle has a `map` method for exactly this purpose:

```
>>> b.map(lambda x: (x.name, set(x.tags)))
[('sequoia', {'huge', 'plant'}),
 ('maple', {'furniture', 'plant', 'syrup'}),
 ('oak', {'building', 'for building', 'plant'}),
 ('elm', {'building', 'firewood', 'paper', 'plant', 'shady'})]
```

This example isn't the most useful, but the point is that we can apply any function across all members without much fanfare, with the results returned in a list and in member order.

The *`map()`* method also features a `processes` parameter, and setting this to an integer greater than 1 will use the `multiprocessing` module internally to map the function across all members using multiple processes. For this to work, we have to give our function an actual name so it can be serialized (pickled) by `multiprocessing`:

```
>>> def get_tags(treant):
...     return (treant.name, set(treant.tags))
>>> b.map(get_tags, processes=2)
[('sequoia', {'huge', 'plant'}),
 ('maple', {'furniture', 'plant', 'syrup'}),
 ('oak', {'building', 'for building', 'plant'}),
 ('elm', {'building', 'firewood', 'paper', 'plant', 'shady'})]
```

For such a simple function and only four Treants in our Bundle, it's unlikely that the parallelism gave any advantage here. But functions that need to do more complicated work with each Treant and the data stored within its tree can gain much from process parallelism when applied to a Bundle of many Treants.

See the API reference for *`map()`* for more details on how to use this method.

4.6.6 API Reference: Bundle

See the *Bundle* API reference for more details.

4.7 Persistent Bundles with Groups

A **Group** is a Treant that can keep track of any number of Treants it counts as members. It can be thought of as having a persistent Bundle attached to it, with that Bundle's membership information stored inside the Group's state file. Since Groups are themselves Treants, they are useful for managing data obtained from a collection of Treants in aggregate.

Note: Since Groups are Treants, everything that applies to Treants applies to Groups as well. They have their own tags, categories, and directory trees, independent of those of their members.

As with a normal Treant, to generate a Group from scratch, we need only give it a name

```
>>> from datreant.core import Group
>>> g = Group('forest')
>>> g
<Group: 'forest'>
```

We can also give any number of existing Treants to add them as members, including other Groups

```
>>> g.members.add('sequoia', 'maple', 'oak', 'elm')
>>> g
<Group: 'forest' | 4 Members>
```

We can access its members directly with:

```
>>> g.members
<MemberBundle([<Treant: 'sequoia'>, <Treant: 'maple'>, <Treant: 'oak'>, <Treant: 'elm'>])>
```

which yields a **MemberBundle**. This object works exactly the same as a Bundle, with the only difference that the order and contents of its membership are stored within the Group's state file. Obtaining subsets of the MemberBundle, such as by slicing, indexing, filtering on tags, etc., will always yield a Bundle:

```
>>> g.members[2:]
<Bundle([<Treant: 'oak'>, <Treant: 'elm'>])>
```

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. Once generated and cached, however, member access will be fast.

A Group can even be a member of itself

```
>>> g.members.add(g)
>>> g
<Group: 'forest' | 5 Members>
>>> g.members[-1]
<Group: 'forest' | 5 Members>
```

4.7.1 Groups find their members when they go missing

For each member Treant, a Group stores the Treant's type ('Treant', 'Group'), its uuid, and last known absolute and relative locations. Since Treants are directories in the filesystem, they can very well be moved. What happens when we load up an old Group whose members may no longer be where they once were?

Upon access to its MemberBundle, a Group will try its best to track down its members. It will first check the last known locations for each member, and for those it has yet to find it will begin a downward search starting from its own tree, then its parent tree, etc. It will continue this process until it either finds all its members, hits the root of the filesystem, or its search times out. If it fails to find a member it will raise an `IOError`.

If the Group you are using fails to find some of its members before timing out, you can set the maximum search time to a longer timeout, in seconds:

```
>>> g.members.searchtime = 60
```

Otherwise, you will need to remove the missing members or find them yourself.

4.7.2 API Reference: Group

See the *Group* API reference for more details.

4.8 API Reference

This is an overview of the `datreant.core` API.

4.8.1 Treants

Treants are the core units of functionality of `datreant`. They function as specially marked directories with distinguishing characteristics. They are designed to be subclassed, with their functionality extendable with attachable Limbs.

The components documented here are those included within `datreant.core`.

Treant

The class *datreant.core.Treant* is the central object of `datreant.core`.

class `datreant.core.`**Treant**(*treant*, *new=False*, *categories=None*, *tags=None*)

The Treant: a Tree with a state file.

treant should be a base directory of a new or existing Treant. An existing Treant will be regenerated if a state file is found. If no state file is found, a new Treant will be created.

A Tree object may also be used in the same way as a directory string.

If multiple Treant state files are in the given directory, a `MultipleTreantsError` will be raised; specify the full path to the desired state file to regenerate the desired Treant in this case. It is generally better to avoid having multiple state files in the same directory.

Use the *new* keyword to force generation of a new Treant at the given path.

Parameters

- **treant** (*str or Tree*) – Base directory of a new or existing Treant; will regenerate a Treant if a state file is found, but will genereate a new one otherwise; may also be a Tree object
- **new** (*bool*) – Generate a new Treant even if one already exists at the given location
- **categories** (*dict*) – dictionary with user-defined keys and values; used to give Treants distinguishing characteristics
- **tags** (*list*) – list with user-defined values; like categories, but useful for adding many distinguishing descriptors

abspath

Absolute path of `self.path`.

attach(**limbname*)

Attach limbs by name to this Treant.

categories

Interface to categories.

children

A View of all files and directories in this Tree.

Includes hidden files and directories.

discover(*dirpath='.'*, *depth=None*, *treantdepth=None*)

Find all Treants within given directory, recursively.

Parameters

- **dirpath** (*string, Tree*) – Directory within which to search for Treants. May also be an existing Tree.
- **depth** (*int*) – Maximum directory depth to tolerate while traversing in search of Treants. `None` indicates no depth limit.
- **treantdepth** (*int*) – Maximum depth of Treants to tolerate while traversing in search of Treants. `None` indicates no Treant depth limit.

Returns **found** – Bundle of found Treants.

Return type *Bundle*

draw(*depth=None*, *hidden=False*)
Print an ASCII-fied visual of the tree.

Parameters

- **depth** (*int*) – Maximum directory depth to display. `None` indicates no limit.
- **hidden** (*bool*) – If False, do not show hidden files; hidden directories are still shown if they contain non-hidden files or directories.

exists
Check existence of this path in filesystem.

filepath
Absolute path to the Treant's state file.

glob(*pattern*)
Return a View of all child Leaves and Trees matching given globbing pattern.

Arguments

pattern globbing pattern to match files and directories with

hidden
A View of the hidden files and directories in this Tree.

leaves
A View of the files in this Tree.

Hidden files are not included.

limbs
A set giving the names of this Tree's attached limbs.

location
The location of the Treant.

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

make()
Make the directory if it doesn't exist. Equivalent to `makedirs()`.

Returns This Tree.

Return type *Tree*

makedirs()
Make all directories along path that do not currently exist.

Returns

tree this tree

name
The name of the Treant.

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

parent
Parent directory for this path.

path
Treant directory as a `pathlib.Path`.

relpath
Relative path of `self.path` from current working directory.

tags
Interface to tags.

treants
Bundle of all Treants found within this Tree.

This does not return a Treant for a bare state file found within this Tree. In effect this gives the same result as `Bundle(self.trees)`.

treanttype
The type of the Treant.

tree
This Treant's directory as a Tree.

trees
A View of the directories in this Tree.

Hidden directories are not included.

uuid
Get Treant uuid.

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

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

has uuid:

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

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

Returns

uuid unique identifier string for this Treant

Tags

The class *datreant.core.limbs.Tags* is the interface used by Treants to access their tags.

class datreant.core.limbs.**Tags**(*treant*)
Interface to tags.

add(**tags*)
Add any number of tags to the Treant.

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

Parameters **tags** (*str or list*) – Tags to add. Must be strings or lists of strings.

clear()
Remove all tags from Treant.

fuzzy(*tag*, *threshold=80*)
Get a tuple of existing tags that fuzzily match a given one.

Parameters

- **tags** (*str or list*) – Tag or tags to get fuzzy matches for.
- **threshold** (*int*) – Lowest match score to return. Setting to 0 will return every tag, while setting to 100 will return only exact matches.

Returns **matches** – Tuple of tags that match.

Return type tuple

remove(**tags*)
Remove tags from Treant.

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

Arguments

tags Tags to delete.

Categories

The class *datreant.core.limbs.Categories* is the interface used by Treants to access their categories.

class datreant.core.limbs.**Categories**(*treant*)
Interface to categories.

add(*categorydict=None*, ***categories*)
Add any number of categories to the Treant.

Categories are key-value pairs that serve to differentiate Treants 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.

Keys must be strings.

Values may be ints, floats, strings, or bools. None as a value will not the existing value for the key, if present.

Parameters

- **categorydict** (*dict*) – Dict of categories to add; keys used as keys, values used as values.
- **categories** (*dict*) – Categories to add. Keyword used as key, value used as value.

clear()
Remove all categories from Treant.

keys()
Get category keys.

Returns

keys keys present among categories

remove(**categories*)
Remove categories from Treant.

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

Parameters **categories** (*str*) – Categories to delete.

values()
Get category values.

Returns

values values present among categories

Group

The class *datreant.core.Group* is a Treant with the ability to store member locations as a persistent Bundle within its state file.

class datreant.core.**Group**(*treant*, *new=False*, *categories=None*, *tags=None*)
A Treant with a persistent Bundle of other Treants.

treant should be a base directory of a new or existing Group. An existing Group will be regenerated if a state file is found. If no state file is found, a new Group will be created.

A Tree object may also be used in the same way as a directory string.

If multiple Treant/Group state files are in the given directory, a `MultipleTreantsError` will be raised; specify the full path to the desired state file to regenerate the desired Group in this case. It is generally better to avoid having multiple state files in the same directory.

Use the *new* keyword to force generation of a new Group at the given path.

Parameters

- **treant** (*str or Tree*) – Base directory of a new or existing Group; will regenerate a Group if a state file is found, but will genereate a new one otherwise; may also be a Tree object
- **new** (*bool*) – Generate a new Group even if one already exists at the given location
- **categories** (*dict*) – dictionary with user-defined keys and values; used to give Groups distinguishing characteristics
- **tags** (*list*) – list with user-defined values; like categories, but useful for adding many distinguishing descriptors

abspath
Absolute path of `self.path`.

attach(**limbname*)
Attach limbs by name to this Treant.

categories
Interface to categories.

children
A View of all files and directories in this Tree.

Includes hidden files and directories.

discover(*dirpath='.'*, *depth=None*, *treantdepth=None*)
Find all Treants within given directory, recursively.

Parameters

- **dirpath** (*string, Tree*) – Directory within which to search for Treants. May also be an existing Tree.
- **depth** (*int*) – Maximum directory depth to tolerate while traversing in search of Treants. `None` indicates no depth limit.
- **treantdepth** (*int*) – Maximum depth of Treants to tolerate while traversing in search of Treants. `None` indicates no Treant depth limit.

Returns **found** – Bundle of found Treants.

Return type *Bundle*

draw(*depth=None*, *hidden=False*)
Print an ASCII-fied visual of the tree.

Parameters

- **depth** (*int*) – Maximum directory depth to display. `None` indicates no limit.
- **hidden** (*bool*) – If False, do not show hidden files; hidden directories are still shown if they contain non-hidden files or directories.

exists
Check existence of this path in filesystem.

filepath
Absolute path to the Treant's state file.

glob(*pattern*)
Return a View of all child Leaves and Trees matching given globbing pattern.

Arguments

pattern globbing pattern to match files and directories with

hidden
A View of the hidden files and directories in this Tree.

leaves
A View of the files in this Tree.

Hidden files are not included.

limbs
A set giving the names of this Tree's attached limbs.

location
The location of the Treant.

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

make()
Make the directory if it doesn't exist. Equivalent to `makedirs()`.

Returns This Tree.

Return type *Tree*

makedirs()
Make all directories along path that do not currently exist.

Returns

tree this tree

members
Persistent Bundle for Groups.

name
The name of the Treant.

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

parent
Parent directory for this path.

path
Treant directory as a `pathlib.Path`.

relpath
Relative path of `self.path` from current working directory.

tags
Interface to tags.

treants
Bundle of all Treants found within this Tree.

This does not return a Treant for a bare state file found within this Tree. In effect this gives the same result as `Bundle(self.trees)`.

treanttype
The type of the Treant.

tree
This Treant's directory as a Tree.

trees
A View of the directories in this Tree.

Hidden directories are not included.

uuid
Get Treant uuid.

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

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

has uuid:

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

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

Returns

uuid unique identifier string for this Treant

Members

The class *datreant.core.limbs.MemberBundle* is the interface used by a Group to manage its members. Its API matches that of *datreant.core.Bundle*.

class datreant.core.limbs.**MemberBundle**(*treant*)
Persistent Bundle for Groups.

abspaths
Return a list of absolute member directory paths.

Members that can't be found will have path None.

Returns

names list giving the absolute directory path of each member, in order; members that are missing will have path None

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

Arguments

treants treants to be added, which may be nested lists of treants; treants can be given as either objects or paths to directories that contain treant statefiles; glob patterns are also allowed, and all found treants will be added to the collection

attach(**agglimbname*)
Attach agglimbs by name to this collection. Attaches corresponding limb to member Treants.

categories
Interface to categories.

clear()
Remove all members.

filepaths
Return a list of member filepaths.

Members that can't be found will have filepath None.

Returns

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

flatten(*exclude=None*)
Return a flattened version of this Bundle.

The resulting Bundle will have all members of any member Groups, without the Groups.

Parameters **exclude** (*list*) – uuids of Groups to leave out of flattening; these will not in the resulting Bundle.

Returns **flattened** – the flattened Bundle with no Groups

Return type *Bundle*

globfilter(*pattern*)
Return a Bundle of members that match by name the given globbing pattern.

Parameters **pattern** (*string*) – globbing pattern to match member names with

limbs
A set giving the names of this collection's attached limbs.

map (*function*, *processes=1*, ***kwargs*)
Apply a function to each member, perhaps in parallel.

A pool of processes is created for *processes* > 1; for example, with 40 members and 'processes=4', 4 processes will be created, each working on a single member at any given time. When each process completes work on a member, it grabs another, until no members remain.

kwargs are passed to the given function when applied to each member

Arguments

function function to apply to each member; must take only a single treant instance as input, but may take any number of keyword arguments

Keywords

processes how many processes to use; if 1, applies function to each member in member order

Returns

results list giving the result of the function for each member, in member order; if the function returns `None` for each member, then only `None` is returned instead of a list

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`

relpaths
Return a list of relative member directory paths.

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

Returns

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

remove (**members*)
Remove any number of members from the collection.

Arguments

members instances or indices of the members to remove

searchtime
Max time to spend searching for missing members, in seconds.

Setting a larger value allows more time for the collection to look for members elsewhere in the filesystem.

If *None*, there will be no time limit. Use with care.

tags
Interface to aggregated tags.

treanttypes
Return a list of member treanttypes.

trees
Obtain a View giving the Tree for each Treant in this Bundle.

uuids
Return a list of member uuids.

Returns

uuids list giving the uuid of each member, in order

4.8.2 Filesystem manipulation

The components of `datreant.core` documented here are those designed for working directly with filesystem objects, namely directories and files.

Tree

The class *datreant.core.Tree* is an interface to a directory in the filesystem.

class `datreant.core.`**Tree**(*dirpath*, *limbs=None*)
A directory.

abspath
Absolute path of `self.path`.

attach(**limbname*)
Attach limbs by name to this Tree.

children
A View of all files and directories in this Tree.

Includes hidden files and directories.

discover(*dirpath='.'*, *depth=None*, *treantdepth=None*)
Find all Treants within given directory, recursively.

Parameters

- **dirpath** (*string, Tree*) – Directory within which to search for Treants. May also be an existing Tree.
- **depth** (*int*) – Maximum directory depth to tolerate while traversing in search of Treants. `None` indicates no depth limit.
- **treantdepth** (*int*) – Maximum depth of Treants to tolerate while traversing in search of Treants. `None` indicates no Treant depth limit.

Returns **found** – Bundle of found Treants.

Return type *Bundle*

draw(*depth=None*, *hidden=False*)
Print an ASCII-fied visual of the tree.

Parameters

- **depth** (*int*) – Maximum directory depth to display. `None` indicates no limit.
- **hidden** (*bool*) – If False, do not show hidden files; hidden directories are still shown if they contain non-hidden files or directories.

exists
Check existence of this path in filesystem.

glob(*pattern*)
Return a View of all child Leaves and Trees matching given globbing pattern.

Arguments

pattern globbing pattern to match files and directories with

hidden
A View of the hidden files and directories in this Tree.

leaves
A View of the files in this Tree.

Hidden files are not included.

limbs
A set giving the names of this Tree's attached limbs.

make()
Make the directory if it doesn't exist. Equivalent to `makedirs()`.

Returns This Tree.

Return type *Tree*

makedirs()
Make all directories along path that do not currently exist.

Returns

tree this tree

name
Basename for this path.

parent
Parent directory for this path.

path
Filesystem path as a `pathlib.Path`.

relpath
Relative path of `self.path` from current working directory.

treants
Bundle of all Treants found within this Tree.

This does not return a Treant for a bare state file found within this Tree. In effect this gives the same result as `Bundle(self.trees)`.

trees
A View of the directories in this Tree.

Hidden directories are not included.

Leaf

The class *datreant.core.Leaf* is an interface to a file in the filesystem.

class datreant.core.**Leaf**(*filepath*)
A file in the filesystem.

abspath
Absolute path.

exists
Check existence of this path in filesystem.

make()
Make the file if it doesn't exit. Equivalent to `touch()`.

Returns

leaf this leaf

makedirs()
Make all directories along path that do not currently exist.

Returns

leaf this leaf

name
Basename for this path.

parent
Parent directory for this path.

path
Filesystem path as a `pathlib.Path`.

read(*size=None*)
Read file, or up to *size* in bytes.

Arguments

size extent of the file to read, in bytes

relpath
Relative path from current working directory.

touch()
Make file if it doesn't exist.

View

The class *datreant.core.View* is an ordered set of Trees and Leaves. It allows for convenient operations on its members as a collective, as well as providing mechanisms for filtering and subselection.

class datreant.core.**View**(**vegs*, ***kwargs*)
An ordered set of Trees and Leaves.

Parameters **vegs** (*Tree, Leaf, or list*) – Trees and/or Leaves to be added, which may be nested lists of Trees and Leaves. Trees and Leaves can be given as either objects or paths.

abspaths
List of absolute paths for the members in this View.

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

Arguments

vegs Trees or Leaves to add; lists, tuples, or other Views with Trees or Leaves will also work; strings giving a path (existing or not) also work, since these are what define Trees and Leaves

attach(**aggtreelimbname*)
Attach aggtreelimbs by name to this View. Attaches corresponding limb to any member Trees.

children
A View of all children within the member Trees.

exists
List giving existence of each member as a boolean.

glob(*pattern*)
Return a View of all child Leaves and Trees of members matching given globbing pattern.

Parameters **pattern** (*string*) – globbing pattern to match files and directories with

globfilter(*pattern*)
Return a View of members that match by name the given globbing pattern.

Parameters **pattern** (*string*) – globbing pattern to match member names with

hidden
A View of the hidden files and directories within the member Trees.

leaves
A View of the files within the member Trees.

Hidden files are not included.

limbs
A set giving the names of this collection's attached limbs.

make()
Make the Trees and Leaves in this View if they don't already exist.

Returns This View.

Return type *View*

map(*function*, *processes=1*, ***kwargs*)
Apply a function to each member, perhaps in parallel.

A pool of processes is created for *processes* > 1; for example, with 40 members and `processes=4`, 4 processes will be created, each working on a single member at any given time. When each process completes work on a member, it grabs another, until no members remain.

kwargs are passed to the given function when applied to each member

Parameters

- **function** (*function*) – Function to apply to each member. Must take only a single Treant instance as input, but may take any number of keyword arguments.
- **processes** (*int*) – How many processes to use. If 1, applies function to each member in member order in serial.

Returns **results** – List giving the result of the function for each member, in member order. If the function returns `None` for each member, then only `None` is returned instead of a list.

Return type list

memberleaves
A View giving only members that are Leaves (or subclasses).

membertrees
A View giving only members that are Trees (or subclasses).

names
List the basenames for the members in this View.

relpaths
List of relative paths from the current working directory for the members in this View.

treants
A Bundle of all existing Treants among the Trees and Leaves in this View.

trees
A View of directories within the member Trees.

Hidden directories are not included.

4.8.3 Treant aggregation

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

Bundle

The class *datreant.core.Bundle* functions as an ordered set of Treants. It allows common operations on Treants to be performed in aggregate, but also includes mechanisms for filtering and grouping based on Treant attributes, such as tags and categories.

Bundles can be created from all Treants found in a directory tree with *datreant.core.discover()*:

`datreant.core.`**discover**(*dirpath='.'*, *depth=None*, *treantdepth=None*)
Find all Treants within given directory, recursively.

Parameters

- **dirpath** (*string, Tree*) – Directory within which to search for Treants. May also be an existing Tree.
- **depth** (*int*) – Maximum directory depth to tolerate while traversing in search of Treants. `None` indicates no depth limit.
- **treantdepth** (*int*) – Maximum depth of Treants to tolerate while traversing in search of Treants. `None` indicates no Treant depth limit.

Returns **found** – Bundle of found Treants.

Return type *Bundle*

They can also be created directly from any number of Treants:

class `datreant.core.`**Bundle**(**treants*, ***kwargs*)
An ordered set of Treants.

Parameters **treants** (*Treant, list*) – Treants to be added, which may be nested lists of Treants. Treants can be given as either objects or paths to directories that contain Treant statefiles. Glob patterns are also allowed, and all found Treants will be added to the collection.

abspaths
Return a list of absolute member directory paths.

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

Returns

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

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

Arguments

treants treants to be added, which may be nested lists of treants; treants can be given as either objects or paths to directories that contain treant statefiles; glob patterns are also allowed, and all found treants will be added to the collection

attach (**agglimbname*)
Attach agglimbs by name to this collection. Attaches corresponding limb to member Treants.

categories
Interface to categories.

clear ()
Remove all members.

filepaths
Return a list of member filepaths.

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

Returns

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

flatten (*exclude=None*)
Return a flattened version of this Bundle.

The resulting Bundle will have all members of any member Groups, without the Groups.

Parameters **exclude** (*list*) – uuids of Groups to leave out of flattening; these will not in the resulting Bundle.

Returns **flattened** – the flattened Bundle with no Groups

Return type *Bundle*

globfilter (*pattern*)
Return a Bundle of members that match by name the given globbing pattern.

Parameters **pattern** (*string*) – globbing pattern to match member names with

limbs
A set giving the names of this collection's attached limbs.

map (*function*, *processes=1*, ***kwargs*)
Apply a function to each member, perhaps in parallel.

A pool of processes is created for *processes* > 1; for example, with 40 members and 'processes=4', 4 processes will be created, each working on a single member at any given time. When each process completes work on a member, it grabs another, until no members remain.

kwargs are passed to the given function when applied to each member

Arguments

function function to apply to each member; must take only a single treant instance as input, but may take any number of keyword arguments

Keywords

processes how many processes to use; if 1, applies function to each member in member order

Returns

results list giving the result of the function for each member, in member order; if the function returns `None` for each member, then only `None` is returned instead of a list

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`

relpaths
Return a list of relative member directory paths.

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

Returns

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

remove (**members*)
Remove any number of members from the collection.

Arguments

members instances or indices of the members to remove

searchtime
Max time to spend searching for missing members, in seconds.

Setting a larger value allows more time for the collection to look for members elsewhere in the filesystem.

If *None*, there will be no time limit. Use with care.

tags
Interface to aggregated tags.

treanttypes
Return a list of member treanttypes.

trees
Obtain a View giving the Tree for each Treant in this Bundle.

uuids
Return a list of member uuids.

Returns

uuids list giving the uuid of each member, in order

AggTags

The class *datreant.core.agglimbs.AggTags* is the interface used by Bundles to access their members' tags.

class datreant.core.agglimbs.**AggTags**(*collection*)
Interface to aggregated tags.

add(**tags*)
Add any number of tags to each Treant in collection.

Arguments

tags Tags to add. Must be strings or lists of strings.

all
Set of tags present among all Treants in collection.

any
Set of tags present among at least one Treant in collection.

clear()
Remove all tags from each Treant in collection.

fuzzy(*tag*, *threshold=80*, *scope='all'*)
Get a tuple of existing tags that fuzzily match a given one.

Parameters

- **tags** (*str or list*) – Tag or tags to get fuzzy matches for.
- **threshold** (*int*) – Lowest match score to return. Setting to 0 will return every tag, while setting to 100 will return only exact matches.
- **scope** (*{'all', 'any'}*) – Tags to use. 'all' will use only tags found within all Treants in collection, while 'any' will use tags found within at least one Treant in collection.

Returns **matches** – Tuple of tags that match.

Return type tuple

remove(**tags*)
Remove tags from each Treant in collection.

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

Arguments

tags Tags to delete.

AggCategories

The class *datreant.core.agglimbs.AggCategories* is the interface used by Bundles to access their members' categories.

class datreant.core.agglimbs.**AggCategories**(*collection*)
Interface to categories.

add(*categorydict=None*, ***categories*)
Add any number of categories to each Treant in collection.

Categories are key-value pairs that serve to differentiate Treants 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.

Keys must be strings.

Values may be ints, floats, strings, or bools. `None` as a value will not the existing value for the key, if present.

Parameters

- **categorydict** (*dict*) – Dict of categories to add; keys used as keys, values used as values.
- **categories** – Categories to add. Keyword used as key, value used as value.

all
Get categories common to all Treants in collection.

Returns Categories common to all members.

Return type dict

any
Get categories present among at least one Treant in collection.

Returns All unique Categories among members.

Return type dict

clear()
Remove all categories from all Treants in collection.

groupby(*keys*)
Return groupings of Treants based on values of Categories.

If a single category is specified by *keys* (*keys* is neither a list nor a set of category names), returns a dict of Bundles whose (new) keys are the values of the category specified by *keys*; the corresponding Bundles are groupings of members in the collection having the same category values (for the category specied by *keys*).

If *keys* is a list or set of keys, returns a dict of Bundles whose (new) keys are tuples of category values. The corresponding Bundles contain the members in the collection that have the same set of category values (for the categories specified by *keys*); members in each Bundle will have all of the category values specified by the tuple for that Bundle's key.

Parameters **keys** (*str, list, set*) – Valid key(s) of categories in this collection.

Returns Bundles of members by category values.

Return type dict

keys(*scope='all'*)
Get the keys present among Treants in collection.

Parameters **scope** (*{'all', 'any'}*) – Keys to return. 'all' will return only keys found within all Treants in the collection, while 'any' will return keys found within at least one Treant in the collection.

Returns **keys** – Present keys.

Return type list

remove(**categories*)
Remove categories from Treant.

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

Parameters **categories** (*str*) – Categories to delete.

values(*scope='all'*)
Get the category values for all Treants in collection.

Parameters **scope** (*{'all', 'any'}*) – Keys to return. 'all' will return only keys found within all Treants in the collection, while 'any' will return keys found within at least one Treant in the collection.

Returns **values** – A list of values for each Treant in the collection is returned for each key within the given *scope*. The value lists are given in the same order as the keys from `AggCategories.keys`.

Return type list

4.9 Contributing to datreant

datreant is an open-source project, with its development driven by the needs of its users. Anyone is welcome to contribute to any of its packages, which can all be found under the datreant GitHub organization.

4.9.1 Development model

datreant subpackages follow the development model outlined by Vincent Driessen , with the `develop` branch being the unstable focal point for development. The `master` branch merges from the develop branch only when all tests are passing, and usually only before a release. In general, master should be usable at all times, while develop may be broken at any particular moment.

4.9.2 Setting up your development environment

We recommend using virtual environments with virtualenvwrapper. Since datreant is a collection of subpackages, you will need to clone whichever repositories you are interested in contributing to. Since all datreant subpackages depend on datreant.core, you will probably want to clone this one:

```
git clone git@github.com:datreant/datreant.core.git
```

Make a new virtualenv called `datreant` with:

```
mkvirtualenv datreant
```

and make a development installation of `datreant.core` with:

```
cd datreant.core
pip install -e .
```

The `-e` flag will cause pip to call setup with the `develop` option. This means that any changes on the source code will immediately be reflected in your virtual environment.

Repeat the same operations for any other datreant subpackage you wish to work on. Note that other subpackages are likely to have heavier dependencies.

4.9.3 Running the tests locally

As you work on a datreant subpackage, it's important to see how your changes affected its expected behavior. With your virtualenv enabled:

```
workon datreant
```

switch to the top-level directory of the subpackage you are working on and run:

```
py.test --cov src/ --pep8 src/
```

This will run all the tests for that subpackage (often inside `src/datreant/<subpackage_name>/tests`), with coverage and PEP8 checks.

Note that to run the tests you will need to install `py.test` and the coverage and PEP8 plugins into your virtualenv:

```
pip install py.test pytest pytest-cov pytest-pep8
```

4.10 Frequently Asked Questions

1. What are some benefits of datreant's approach to data management?

   It's daemonless. There is no server process required on the machine to read/write persistent objects. This is valuable when working with data on remote resources, where it might not be possible to set up one's own daemon to talk to.

   Treants are portable. Because they store their state in JSON, and because treants store their data in the filesystem, they are easy to move around piecemeal. If you want to use a treant on a remote system, but don't want to drag all its stored datasets with it, you can copy only what you need.

   Contrast this with many database solutions, in which you either copy the whole database somehow, or slurp the pieces of data out that you want. Most database solutions can be rather slow to do this.

   Treants are independent. Although Groups are aware of their members, Treants work independently from one another. If you want to use only basic Treants, that works just fine. If you want to use Groups, that works, too.

   Treants have a structure in the filesystem. This means that all the shell tools we know and love are available to work with their contents, which might include plaintext files, figures, topology files, simulation trajectories, random pickles, ipython notebooks, html files, etc. Basically, treants are as versatile as the filesystem is, at least when it comes to storage.

2. What are some disadvantages of datreant's design?

   Treants could be anywhere in the filesystem. This is mostly a problem for Groups, which allow aggregation of other Treants. If a member is moved, the Group has no way of knowing where it went; we've built machinery to help it find its members, but these will always be limited to filesystem search methods (some quite good, but still). If these objects lived in a single database, this wouldn't be an issue.

   Queries on object metadata will be slower than a central database. We want Groups and Bundles (in-memory Groups, basically) to be able to run queries against their members' characteristics, returning subsets matching the query. Since these queries have to be applied against these objects and not against a single table somewhere, it will be relatively slow.

   File locking is less efficient for multiple-read/write under load than a smart daemon process/scheduler. The assumption we make is that Treants are primarily read, and only occasionally written to. This is assumed for their data and their metadata. They are not designed to scale well if the same parts are being written to and read at the same time by many processes.

   Having Treants exist as separate files (state files and data all separate) does mitigate this potential for gridlock, which is one reason we favor many files over few. But it's still something to be aware of.

A

B

C

D

E

F

G

U

V