datreant Documentation

Release 1.0.2

David Dotson

Jun 29, 2018

User Documentation

1 Stay organized 3

2 Getting datreant 5

3 Contributing 7

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 Python package that provides 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

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.git
```

CHAPTER 3

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.

3.1 Installing datreant

You can install `datreant` from PyPI using pip:

```
pip install datreant
```

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

```
pip install --user datreant
```

Alternatively we also provide conda package.

```
conda install -c datreant datreant
```

All datreant packages currently support the following Python versions:

```
- 2.7
- 3.3
- 3.4
- 3.5
- 3.6
```

3.1.1 Dependencies

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

```
- asciitree
- pathlib
- scandir
- six
- fuzzywuzzy
- python-Levenshtein
```

These are automatically installed when installing `datreant`.

3.1.2 Installing from source

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

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

3.2 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.

3.2.1 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 as dtr
>>> s = dtr.Treant('sprout')
```

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

```
> # shell
> ls -a sprout
.  ..  .datreant
```

This `.datreant` directory is what makes `sprout` a Treant. On its own it serves as a marker, but as we'll see later it can also contain metadata elements distinguishing this Treant from others.

Treants are persistent. In fact, we can open a separate python session (go ahead!) and use this Treant immediately there

```
>>> # python session 2
>>> import datreant 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 place on disk; they store almost nothing in memory.

Note: File handles are kept alive only as long as needed to serialize or deserialize Treant metadata from the filesystem.

3.2.2 API Reference: Treant

See the *Treant* API reference for more details.

3.3 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.

3.3.1 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.

3.3.2 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.

3.3.3 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.

3.4 Filesystem manipulation with Trees and Leaves

A Treant functions as a specially marked directory, containing a `.datreant` directory possibly with identifying information inside. What's a Treant without a `.datreant` directory? 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 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.

3.4.1 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.

If you don't want to rely on ending `/` characters when making new directories, we can use *`treeloc()`* to guarantee it:

```
>>> adir = t.treeloc['a/directory'].make()  # will always make a directory
```

and likewise for files, to be explicit:

```
>>> afile = t.leafloc['a/file'].make()      # will always make a file
```

Synchronizing Trees

Synchronization of Tree contents can be performed through the *`sync()`* method. Synchronization can be performed both locally and remotely, and is done through the rsync command:

```
>>> sequoia = dtr.Tree('sequoia')
>>> oak = dtr.Tree('oak')
>>> sequoia.sync(oak, mode="download")  # Sync contents from oak to sequoia
>>> sequoia.sync("/tmp/sequoia", mode="upload")  # Sync to a local directory
>>> sequoia.sync("user@host:/directory")  # Sync remotely
```

Note: To be able to sync remotely, it is necessary to have passwordless ssh access (through key file) to the server.

API Reference: Tree

See the *Tree* API reference for more details.

3.4.2 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(hidden=True)
sprout/
 +-- .datreant/
 +-- a/
     +-- .hidden/
     |   +-- directory/
     +-- new/
         +-- file
```

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(['.hidden/', '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*.

3.4.3 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.

3.5 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 immutable, ordered set of its members.

3.5.1 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 as dtr
>>> import glob
>>> v = dtr.View(glob.glob('*'))
>>> v
<View(['moe/', 'larry/', '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(['larry/', 'curly.txt'])>
```

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

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

Or boolean indexing

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

As well as indexing by name

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

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

3.5.2 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(['moe/', 'larry/'])>
```

Or only the Leaves?

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

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

```
>>> v.globfilter('*r*')
<View(['larry/', 'curly.txt'])>
```

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

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

3.5.3 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(['about_moe.txt', 'more_moe.pdf'])>
```

and the directories with

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

Both these properties leave out hidden children by default, 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.trees(hidden=True)
<View(['.hiding_here/', 'sprout/'])>
```

Want all the children?

```
>>> moe.children(hidden=True)
<View(['sprout/', 'about_moe.txt', 'more_moe.pdf', '.hiding_here/'])>
```

3.5.4 A View is an ordered set

Because a View is a set, adding members that are already present results in no a new View with nothing additional:

```
>>> v = v + Tree('moe')
>>> v
<View(['moe/', 'larry/', '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(['curly.txt', 'larry/', '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(['moe/', 'larry/', 'curly.txt', 'nonexistent_file.txt'])>
```

the intersection:

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

differences:

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

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

or the symmetric difference:

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

3.5.5 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(hidden=True)
<View(['sprout/', 'about_moe.txt', 'more_moe.pdf', '.hiding_here',
       'about_larry.txt'])>
```

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

```
>>> v.glob('*moe*')
<View(['about_moe.txt', '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.

3.5.6 API Reference: View

See the *View* API reference for more details.

3.6 Coordinating Treants with Bundles

Similar to a View, a **Bundle** is an immutable, 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 as dtr
>>> b = dtr.Bundle('elm', 'maple', 'oak', 'sequoia')
>>> b
<Bundle(['elm', 'maple', 'oak', '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.

3.6.1 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(['sequoia', 'maple', 'oak', '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.discover()* API reference for more details.

3.6.2 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(['maple', 'oak', 'elm'])>
```

fancy indexing:

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

boolean indexing:

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

and indexing by Treant name:

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

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

3.6.3 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.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(['oak', 'elm'])>
```

We can also provide multiple tags to match more Treants:

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

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

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

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

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

3.6.4 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.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'],
```

(continues on next page)

(continued from previous page)

```
                '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(['maple', 'oak', 'elm'])>,
 'evergreen': <Bundle(['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(['maple', 'oak', 'elm'])>
```

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

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

3.6.5 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.

3.6.6 API Reference: Bundle

See the *Bundle* API reference for more details.

3.7 API Reference

This is an overview of the datreant API.

3.7.1 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.

Treant

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

class datreant.**Treant**(*treant*, *categories=None*, *tags=None*)

The Treant: a discoverable Tree with metadata.

treant should be the directory of a new or existing Treant. An existing Treant will be used if a *.datreant* directory is found inside. If no *.datreant* directory is found inside, a new Treant will be created.

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

Parameters

- **treant** (*str or Tree*) – Base directory of a new or existing Treant; may also be a Tree object
- **categories** (*dict*) – dictionary with user-defined keys and values; used to give Treants distinguishing characteristics
- **tags** (*list*) – list with user-defined strings; like categories, but useful for adding many distinguishing descriptors

abspath

Absolute path of self.path.

children(*hidden=False*)

Return a View of all files and directories in this Tree.

Parameters **hidden** (*bool*) – If True, include hidden files and directories.

Returns A View with files and directories in this Tree as members.

Return type *View*

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

Parameters

- **depth** (*int, optional*) – Maximum directory depth to display. `None` indicates no limit.
- **hidden** (*bool*) – If True, show hidden files and directories.

exists
Check existence of this path in filesystem.

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

Parameters **pattern** – globbing pattern to match files and directories with

leafloc
Get Leaf at relative *path*.

Use with getitem syntax, e.g. `.treeloc['some name']`

Allowed inputs are: - A single name - A list or array of names

If the given path resolves to an existing directory, then a `ValueError` will be raised.

leaves(*hidden=False*)
Return a View of the files in this Tree.

Parameters **hidden** (*bool*) – If True, include hidden files.

Returns A View with files in this Tree as members.

Return type *View*

loc
Get Tree/Leaf at relative *path*.

Use with getitem syntax, e.g. `.loc['some name']`

Allowed inputs are: - A single name - A list or array of names

If directory/file does not exist at the given path, then whether a Tree or Leaf is given is determined by the path semantics, i.e. a trailing separator ("/").

Using e.g. `Tree.loc['some name']` is equivalent to doing `Tree['some name']`. `.loc` is included for parity with `View` and `Bundle` API semantics.

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 This Tree.

Return type *Tree*

name
The name of the Treant.

parent
Parent directory for this path.

path
Filesystem path as a `pathlib2.Path`.

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

sync(*other*, *mode='upload'*, *compress=True*, *checksum=True*, *backup=False*, *dry=False*, *include=None*, *exclude=None*, *overwrite=False*, *rsync_path='/usr/bin/rsync'*)
Synchronize directories using rsync.

Parameters

- **other** (*str or Tree*) – Other end of the sync, can be either a path or another Tree.
- **mode** (*str*) – Either `"upload"` if uploading to *other*, or `"download"` if downloading from *other*

The other options are described in the *datreant.rsync.rsync()* documentation.

treeloc
Get Tree at relative *path*.

Use with getitem syntax, e.g. `.treeloc['some name']`

Allowed inputs are: - A single name - A list or array of names

If the given path resolves to an existing file, then a `ValueError` will be raised.

trees(*hidden=False*)
Return a View of the directories in this Tree.

Parameters **hidden** (*bool*) – If True, include hidden directories.

Returns A View with directories in this Tree as members.

Return type *View*

walk(*topdown=True*, *onerror=None*, *followlinks=False*)
Walk through the contents of the tree.

For each directory in the tree (including the root itself), yields a 3-tuple (dirpath, dirnames, filenames).

Parameters

- **topdown** (*Boolean, optional*) – If False, walks directories from the bottom-up.
- **onerror** (*function, optional*) – Optional function to be called on error.
- **followlinks** (*Boolean, optional*) – If False, excludes symbolic file links.

Returns Wrapped *scandir.walk()* generator yielding *datreant* objects

Return type generator

Tags

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

class datreant.metadata.**Tags**(*tree*)
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.metadata.Categories* is the interface used by Treants to access their categories.

class datreant.metadata.**Categories**(*tree*)
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 delete the existing value for the key, if present, and is otherwise not allowed.

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

3.7.2 Filesystem manipulation

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

Tree

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

class datreant.**Tree**(*dirpath*)
A directory.

abspath
Absolute path of `self.path`.

children(*hidden=False*)
Return a View of all files and directories in this Tree.

Parameters **hidden** (*bool*) – If True, include hidden files and directories.

Returns A View with files and directories in this Tree as members.

Return type *View*

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

Parameters

- **depth** (*int, optional*) – Maximum directory depth to display. `None` indicates no limit.
- **hidden** (*bool*) – If True, show hidden files and directories.

exists
Check existence of this path in filesystem.

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

Parameters **pattern** – globbing pattern to match files and directories with

leafloc
Get Leaf at relative *path*.

Use with getitem syntax, e.g. `.treeloc['some name']`

Allowed inputs are: - A single name - A list or array of names

If the given path resolves to an existing directory, then a `ValueError` will be raised.

leaves(*hidden=False*)
Return a View of the files in this Tree.

Parameters **hidden** (*bool*) – If True, include hidden files.

Returns A View with files in this Tree as members.

Return type *View*

loc
Get Tree/Leaf at relative *path*.

Use with getitem syntax, e.g. `.loc['some name']`

Allowed inputs are: - A single name - A list or array of names

If directory/file does not exist at the given path, then whether a Tree or Leaf is given is determined by the path semantics, i.e. a trailing separator ("/").

Using e.g. `Tree.loc['some name']` is equivalent to doing `Tree['some name']`. `.loc` is included for parity with `View` and `Bundle` API semantics.

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 This Tree.

Return type *Tree*

name
Basename for this path.

parent
Parent directory for this path.

path
Filesystem path as a `pathlib2.Path`.

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

sync(*other*, *mode='upload'*, *compress=True*, *checksum=True*, *backup=False*, *dry=False*, *include=None*, *exclude=None*, *overwrite=False*, *rsync_path='/usr/bin/rsync'*)
Synchronize directories using rsync.

Parameters

- **other** (*str or Tree*) – Other end of the sync, can be either a path or another Tree.
- **mode** (*str*) – Either `"upload"` if uploading to *other*, or `"download"` if downloading from *other*

The other options are described in the *datreant.rsync.rsync()* documentation.

treeloc
Get Tree at relative *path*.

Use with getitem syntax, e.g. `.treeloc['some name']`

Allowed inputs are: - A single name - A list or array of names

If the given path resolves to an existing file, then a `ValueError` will be raised.

trees(*hidden=False*)
Return a View of the directories in this Tree.

Parameters **hidden** (*bool*) – If True, include hidden directories.

Returns A View with directories in this Tree as members.

Return type *View*

walk(*topdown=True*, *onerror=None*, *followlinks=False*)
Walk through the contents of the tree.

For each directory in the tree (including the root itself), yields a 3-tuple (dirpath, dirnames, filenames).

Parameters

- **topdown** (*Boolean, optional*) – If False, walks directories from the bottom-up.
- **onerror** (*function, optional*) – Optional function to be called on error.
- **followlinks** (*Boolean, optional*) – If False, excludes symbolic file links.

Returns Wrapped *scandir.walk()* generator yielding *datreant* objects

Return type generator

Leaf

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

class datreant.**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

Return type *Leaf*

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

Returns **leaf** – this leaf

Return type *Leaf*

name
Basename for this path.

parent
Parent directory for this path.

path
Filesystem path as a `pathlib2.Path`.

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

Parameters **size** (*int*) – 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.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.**View**(**vegs*)
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.

children(*hidden=False*)
Return a View of all files and directories within the member Trees.

Parameters **hidden** (*bool*) – If True, include hidden files and directories.

Returns A View giving the files and directories in the member Trees.

Return type *View*

draw(*depth=None*, *hidden=False*)
Print an ASCII-fied visual of all member Trees.

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
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

leafloc
Get a View giving Leaf at *path* relative to each Tree in collection.

Use with getitem syntax, e.g. `.loc['some name']`

Allowed inputs are: - A single name - A list or array of names

If the given path resolves to an existing directory for any Tree, then a `ValueError` will be raised.

leaves (*hidden=False*)
Return a View of the files within the member Trees.

Parameters **hidden** (*bool*) – If True, include hidden files.

Returns A View giving the files in the member Trees.

Return type *View*

loc
Get a View giving Tree/Leaf at *path* relative to each Tree in collection.

Use with getitem syntax, e.g. `.loc['some name']`

Allowed inputs are: - A single name - A list or array of names

If directory/file does not exist at the given path, then whether a Tree or Leaf is given is determined by the path semantics, i.e. a trailing separator ("/").

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.

parents()
Return a View of the parent directories for each member.

Because a View functions as an ordered set, and some members of this collection may share a parent, the View of parents may contain fewer elements than this collection.

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

treeloc
Get a View giving Tree at *path* relative to each Tree in collection.

Use with getitem syntax, e.g. `.loc['some name']`

Allowed inputs are: - A single name - A list or array of names

If the given path resolves to an existing file for any Tree, then a `ValueError` will be raised.

trees(*hidden=False*)
Return a View of the directories within the member Trees.

Parameters **hidden** (*bool*) – If True, include hidden directories.

Returns A View giving the directories in the member Trees.

Return type *View*

3.7.3 Treant aggregation

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

Bundle

The class *datreant.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.discover()*:

datreant.**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.**Bundle**(**treants*)
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.

Returns

abspaths list giving the absolute directory path of each member, in order

children (*hidden=False*)
Return a View of all files and directories within the member Trees.

Parameters **hidden** (*bool*) – If True, include hidden files and directories.

Returns A View giving the files and directories in the member Trees.

Return type *View*

draw (*depth=None*, *hidden=False*)
Print an ASCII-fied visual of all member Trees.

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.

get (**tags*, ***categories*)
Filter to only Treants which match the defined tags and categories.

If no arguments given, the full Bundle is returned. This method should be thought of as a filtering, with more values specified giving only those Treants that match.

Parameters

- ***tags** – Tags to match.
- ****categories** – Category key, value pairs to match.

Returns All matched Treants.

Return type *Bundle*

Examples

Doing a *get* with:

```
>>> b.get('this')
```

is equivalent to:

```
>>> b.tags.filter('this')
```

Finally, doing:

```
>>> b.get('this', length=5)
```

is equivalent to:

```
>>> b_n = b.tags.filter('this')
>>> b_n.categories.groupby('length')[5.0]
```

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 Bundle of members that match by name the given globbing pattern.

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

leafloc
Get a View giving Leaf at *path* relative to each Tree in collection.

Use with getitem syntax, e.g. `.loc['some name']`

Allowed inputs are: - A single name - A list or array of names

If the given path resolves to an existing directory for any Tree, then a `ValueError` will be raised.

leaves (*hidden=False*)
Return a View of the files within the member Trees.

Parameters **hidden** (*bool*) – If True, include hidden files.

Returns A View giving the files in the member Trees.

Return type *View*

loc
Get a View giving Tree/Leaf at *path* relative to each Tree in collection.

Use with getitem syntax, e.g. `.loc['some name']`

Allowed inputs are: - A single name - A list or array of names

If directory/file does not exist at the given path, then whether a Tree or Leaf is given is determined by the path semantics, i.e. a trailing separator ("/").

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.

Returns

names list giving the name of each member, in order

parents()
Return a View of the parent directories for each member.

Because a View functions as an ordered set, and some members of this collection may share a parent, the View of parents may contain fewer elements than this collection.

relpaths
Return a list of relative member directory paths.

Returns

names list giving the relative directory path of each member, in order

treeloc
Get a View giving Tree at *path* relative to each Tree in collection.

Use with getitem syntax, e.g. `.loc['some name']`

Allowed inputs are: - A single name - A list or array of names

If the given path resolves to an existing file for any Tree, then a `ValueError` will be raised.

trees(*hidden=False*)
Return a View of the directories within the member Trees.

Parameters **hidden** (*bool*) – If True, include hidden directories.

Returns A View giving the directories in the member Trees.

Return type *View*

AggTags

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

class datreant.metadata.**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.

filter(*tag*)
Filter Treants matching the given tag expression from a Bundle.

Parameters **tag** (*str or list*) – Tag or tags to filter Treants.

Returns Bundle of Treants matching the given tag expression.

Return type *Bundle*

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

Parameters

- **tag** (*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.metadata.AggCategories* is the interface used by Bundles to access their members' categories.

class datreant.metadata.**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 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*) – 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

3.7.4 Treant synchronization

These are the API components of `datreant` for synchronizing Treants locally and remotely.

Sync

The synchronization functionality is provided by the rsync wrapper function. The function is used by the *datreant.Tree.sync()* method.

datreant.rsync.**rsync**(*source*, *dest*, *compress=True*, *backup=False*, *dry=False*, *checksum=True*, *include=None*, *exclude=None*, *overwrite=False*, *rsync_path='/usr/bin/rsync'*)

Wrapper function for rsync. There are some minor differences with the standard rsync behaviour:

- The include option will exclude every other path.
- The exclude statements will take place after the include statements

Parameters

- **source** (*str*) – Source directory for the sync
- **dest** (*str*) – Dest directory for the sync
- **compress** (*bool*) – If True, use gzip compression to reduce the data transfered over the network
- **backup** (*bool*) – If True, pre-existing files are renamed with a "~" extension before they are replaced
- **dry** (*bool*) – If True, do a dry-run. Useful for debugging purposes
- **checksum** (*bool*) – Perform a checksum to determine if file content has changed.
- **include** (*str or list*) – Paths (wildcards are allowed) to be included. If this option is used, every other path is excluded
- **exclude** (*str or list*) – Paths to be excluded from the copy
- **overwrite** (*bool*) – If False, files in *dest* that are newer than files in *source* will not be overwritten
- **rsync_path** (*str*) – Path where to find the rsync executable

3.8 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 the project, which centers around the datreant GitHub organization.

3.8.1 Development model

datreant follows 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.

3.8.2 Setting up your development environment

We recommend using virtual environments with virtualenvwrapper. First, clone the repository:

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

Make a new virtualenv called `datreant` with:

```
mkvirtualenv datreant
```

and make a development installation of `datreant` with:

```
cd datreant
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.

3.8.3 Running the tests locally

As you work on datreant, 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 package and run:

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

This will run all the tests (inside `src/datreant/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 pytest pytest-cov pytest-pep8
```

Index

A

abspath (datreant.Leaf attribute), 29
abspath (datreant.Treant attribute), 23
abspath (datreant.Tree attribute), 27
abspaths (datreant.Bundle attribute), 33
abspaths (datreant.View attribute), 30
add() (datreant.metadata.AggCategories method), 36
add() (datreant.metadata.AggTags method), 35
add() (datreant.metadata.Categories method), 26
add() (datreant.metadata.Tags method), 25
AggCategories (class in datreant.metadata), 36
AggTags (class in datreant.metadata), 35
all (datreant.metadata.AggCategories attribute), 36
all (datreant.metadata.AggTags attribute), 35
any (datreant.metadata.AggCategories attribute), 36
any (datreant.metadata.AggTags attribute), 35

B

Bundle (class in datreant), 32

C

Categories (class in datreant.metadata), 26
children() (datreant.Bundle method), 33
children() (datreant.Treant method), 23
children() (datreant.Tree method), 27
children() (datreant.View method), 30
clear() (datreant.metadata.AggCategories method), 37
clear() (datreant.metadata.AggTags method), 35
clear() (datreant.metadata.Categories method), 26
clear() (datreant.metadata.Tags method), 26

D

discover() (in module datreant), 32
draw() (datreant.Bundle method), 33
draw() (datreant.Treant method), 24
draw() (datreant.Tree method), 27
draw() (datreant.View method), 30

E

exists (datreant.Leaf attribute), 29
exists (datreant.Treant attribute), 24
exists (datreant.Tree attribute), 27
exists (datreant.View attribute), 30

F

filter() (datreant.metadata.AggTags method), 35
fuzzy() (datreant.metadata.AggTags method), 36
fuzzy() (datreant.metadata.Tags method), 26

G

get() (datreant.Bundle method), 33
glob() (datreant.Bundle method), 34
glob() (datreant.Treant method), 24
glob() (datreant.Tree method), 27
glob() (datreant.View method), 30
globfilter() (datreant.Bundle method), 34
globfilter() (datreant.View method), 30
groupby() (datreant.metadata.AggCategories method), 37

K

keys() (datreant.metadata.AggCategories method), 37
keys() (datreant.metadata.Categories method), 26

L

Leaf (class in datreant), 29
leafloc (datreant.Bundle attribute), 34
leafloc (datreant.Treant attribute), 24
leafloc (datreant.Tree attribute), 27
leafloc (datreant.View attribute), 30
leaves() (datreant.Bundle method), 34
leaves() (datreant.Treant method), 24
leaves() (datreant.Tree method), 28
leaves() (datreant.View method), 31
loc (datreant.Bundle attribute), 34
loc (datreant.Treant attribute), 24
loc (datreant.Tree attribute), 28
loc (datreant.View attribute), 31

M

make() (datreant.Leaf method), 29

make() (datreant.Treant method), 24
make() (datreant.Tree method), 28
make() (datreant.View method), 31
makedirs() (datreant.Leaf method), 29
makedirs() (datreant.Treant method), 24
makedirs() (datreant.Tree method), 28
map() (datreant.Bundle method), 34
map() (datreant.View method), 31
memberleaves (datreant.View attribute), 31
membertrees (datreant.View attribute), 31

N

name (datreant.Leaf attribute), 29
name (datreant.Treant attribute), 24
name (datreant.Tree attribute), 28
names (datreant.Bundle attribute), 34
names (datreant.View attribute), 31

P

parent (datreant.Leaf attribute), 29
parent (datreant.Treant attribute), 25
parent (datreant.Tree attribute), 28
parents() (datreant.Bundle method), 35
parents() (datreant.View method), 31
path (datreant.Leaf attribute), 30
path (datreant.Treant attribute), 25
path (datreant.Tree attribute), 28

R

read() (datreant.Leaf method), 30
relpath (datreant.Leaf attribute), 30
relpath (datreant.Treant attribute), 25
relpath (datreant.Tree attribute), 28
relpaths (datreant.Bundle attribute), 35
relpaths (datreant.View attribute), 32
remove() (datreant.metadata.AggCategories method), 37
remove() (datreant.metadata.AggTags method), 36
remove() (datreant.metadata.Categories method), 27
remove() (datreant.metadata.Tags method), 26
rsync() (in module datreant.rsync), 38

S

sync() (datreant.Treant method), 25
sync() (datreant.Tree method), 28

T

Tags (class in datreant.metadata), 25
touch() (datreant.Leaf method), 30
Treant (class in datreant), 23
Tree (class in datreant), 27
treeloc (datreant.Bundle attribute), 35
treeloc (datreant.Treant attribute), 25
treeloc (datreant.Tree attribute), 29
treeloc (datreant.View attribute), 32
trees() (datreant.Bundle method), 35
trees() (datreant.Treant method), 25
trees() (datreant.Tree method), 29
trees() (datreant.View method), 32

V

values() (datreant.metadata.AggCategories method), 37
values() (datreant.metadata.Categories method), 27
View (class in datreant), 30

W

walk() (datreant.Treant method), 25
walk() (datreant.Tree method), 29