ytree Documentation

Release 2.0.1

Britton Smith

Nov 14, 2017

Contents

ytree is a tool for working with merger-tree data from multiple sources. ytree is an extension of the yt analysis toolkit and provides a similar interface for merger-tree data that includes universal field names, derived fields, and symbolic units. ytree can create merger-trees from Gadget FoF/Subfind catalogs, either for all halos or for a specific set of halos. ytree is able to load in merger-tree from the following formats:

- consistent-trees
- Rockstar halo catalogs without consistent-trees
- *merger-trees made with ytree*

All formats can be saved with a universal format that can be reloaded with ytree. Individual trees for single halos can also be saved.

CHAPTER 1

Table of Contents

1.1 Installation

ytree's main dependency is yt. Once you have installed `yt` following the instructions here, ytree can be installed using pip.

```
$ pip install ytree
```

If you'd like to install the development version, the repository can be found at https://github.com/brittonsmith/ytree. This can be installed by doing:

```
$ git clone https://github.com/brittonsmith/ytree
$ cd ytree
$ pip install -e .
```

1.2 What version do I have?

To see what version of ytree you are using, do the following:

```
import ytree
print (ytree.__version__)
```

1.3 Sample Data

Sample datasets for every supported data format are available for download from the yt Hub in the ytree data collection. The entire collection (only about 33 MB) can be downloaded via the yt Hub's web interface by clicking on "Actions" drop-down menu on the far right and selecting "Download collection." It can also be downloaded through the girder-client interface:

```
$ pip install girder-client
$ girder-cli --api-url https://girder.hub.yt/api/v1 download 59835a1ee2a67400016a2cda␣
↪ytree_data
```

1.4 Working with Merger-Trees

The *Arbor* class is responsible for loading and providing access to merger-tree data. Below, we demonstrate how to load data and what can be done with it.

1.4.1 Loading Merger-Tree Data

ytree can load merger-tree data from multiple sources using the `load` command.

```
import ytree
a = ytree.load("consistent_trees/tree_0_0_0.dat")
```

This command will determine the correct format and read in the data accordingly. For examples of loading each format, see below.

Loading Data

Below are instructions for loading all supported datasets.

Consistent-Trees

The consistent-trees format is typically one or a few files with a naming convention like "tree_0_0_0.dat". To load these files, just give the filename

```
import ytree
a = ytree.load("consistent_trees/tree_0_0_0.dat")
```

Rockstar Catalogs

Rockstar catalogs with the naming convention "out_*.list" will contain information on the descendent ID of each halo and can be loaded independently of consistent-trees. This can be useful when your simulation has very few halos, such as in a zoom-in simulation. To load in this format, simply provide the path to one of these files.

```
import ytree
a = ytree.load("rockstar/rockstar_halos/out_0.list")
```

TreeFarm

Merger-trees created with *TreeFarm* (ytree's merger-tree code for Gadget FoF/SUBFIND catalogs) can be loaded in by providing the path to one of the catalogs created during the calculation.

```
import ytree
a = ytree.load("tree_farm/tree_farm_descendents/fof_subhalo_tab_000.0.h5")
```

Saved Arbors

Once merger-tree data has been loaded, it can be saved to a universal format using *save_arbor* or *save_tree*. These can be loaded by providing the path to the primary hdf5 file.

```
import ytree
a = ytree.load("arbor/arbor.h5")
```

Saved Arbors from ytree 1.1

Arbors created with version 1.1 of ytree and earlier can be reloaded by providing the single file created. It is recommended that arbors be re-saved into the newer format as this will significantly improve performance.

```
import ytree
a = ytree.load("arbor.h5")
```

1.4.2 Working with Merger-Tree Data

Very little happens immediately after a dataset has been loaded. All tree construction and data access occurs only on demand. After loading, information such as the simulation box size, cosmological parameters, and the available fields can be accessed.

```
>>> print (a.box_size)
100.0 Mpc/h
>>> print (a.hubble_constant, a.omega_matter, a.omega_lambda)
0.695 0.285 0.715
>>> print (a.field_list)
['scale', 'id', 'desc_scale', 'desc_id', 'num_prog', ...]
```

Similar to yt, ytree supports accessing fields by their native names as well as generalized aliases. For more information on fields in ytree, see *Fields in ytree*.

How many trees are there?

As soon as any information about the collection of trees within the loaded dataset is requested, an array will be constructed containing objects representing the root of each tree, i.e., the last descendent halo. This structure is accessed by querying the loaded `Arbor` directly. It can also be accessed as `a.trees`.

```
>>> print (a.size)
Loading tree roots: 100%|| 5105985/5105985 [00:00<00:00, 505656111.95it/s]
327
```

Root Fields

Field data for all tree roots is accessed by querying the `Arbor` in a dictionary-like manner.

```
>>> print (a["mass"])
Getting root fields: 100%|| 327/327 [00:00<00:00, 9108.67it/s]
[  6.57410072e+14   5.28489209e+14   5.18129496e+14   4.88920863e+14, ...,
   8.68489209e+11   8.68489209e+11   8.68489209e+11] Msun
```

ytree uses yt's system for symbolic units, allowing for simple unit conversion.

```
>>> print (a["virial_radius"].to("Mpc/h"))
[ 1.583027  1.471894  1.462154  1.434253  1.354779  1.341322  1.28617, ...,
  0.173696  0.173696  0.173696  0.173696  0.173696] Mpc/h
```

When dealing with cosmological simulations, care must be taken to distinguish between comoving and proper reference frames. Please read *An Important Note on Comoving and Proper Units* before your magical ytree journey begins.

Accessing Individual Trees

Individual trees can be accessed by indexing the `Arbor` object.

```
>>> print (a[0])
TreeNode[12900]
```

A *TreeNode* is one halo in a merger-tree. The number is the universal identifier associated with halo. It is unique to the whole arbor. Fields can be accessed for any given `TreeNode` in the same dictionary-like fashion.

```
>>> print (a[0]["mass"])
657410071942446.1 Msun
```

The full lineage of the tree can be accessed by querying any `TreeNode` with the *tree* keyword.

```
>>> my_tree = a[0]
>>> print (my_tree["tree"])
[TreeNode[12900] TreeNode[12539] TreeNode[12166] TreeNode[11796] ...
 TreeNode[591]]
```

Fields can be queried for the tree by including the field name.

```
>>> print (my_tree["tree", "virial_radius"])
[ 2277.73669065  2290.65899281  2301.43165468  2311.47625899  2313.99280576 ...
  434.59856115   410.13381295   411.25755396] kpc
```

A halo's ancestors are stored as a list in the `ancestors` attribute. The descendents are stored in a similar fashion.

```
>>> print (my_tree.ancestors)
[TreeNode[12539]]
>>> print (my_tree.ancestors[0].descendent)
TreeNode[12900]
```

Accessing the Progenitor Lineage of a Tree

Similar to the *tree* keyword, the *prog* keyword can be used to access the line of main progenitors.

```
>>> print (my_tree["prog"])
[TreeNode[12900] TreeNode[12539] TreeNode[12166] TreeNode[11796] ...
 TreeNode[62]]
>>> print (my_tree["prog", "mass"])
[  6.57410072e+14   6.57410072e+14   6.53956835e+14   6.50071942e+14 ...
   8.29496403e+13   7.72949640e+13   6.81726619e+13   5.99280576e+13] Msun
```

Customizing the Progenitor Line

By default, the progenitor line is defined as the line of the most massive ancestors. This can be changed by calling the *set_selector*.

```
>>> a.set_selector("max_field_value", "virial_radius")
```

New selector functions can also be supplied. These functions should minimally accept a list of ancestors and return a single `TreeNode`.

```
>>> def max_value(ancestors, field):
...     vals = np.array([a[field] for a in ancestors])
...     return ancestors[np.argmax(vals)]
...
>>> ytree.add_tree_node_selector("max_field_value", max_value)
>>>
>>> a.set_selector("max_field_value", "mass")
>>> print (a[0]["prog"])
```

1.4.3 Saving Arbors and Trees

`Arbors` of any type can be saved to a universal file format with the *save_arbor* function. These can be reloaded with the *load* command. This format is optimized for fast tree-building and field-access and so is recommended for most situations.

```
>>> fn = a.save_arbor()
Setting up trees: 100%|| 327/327 [00:00<00:00, 483787.45it/s]
Getting fields [1/1]: 100%|| 327/327 [00:00<00:00, 36704.51it/s]
Creating field arrays [1/1]: 100%|| 613895/613895 [00:00<00:00, 7931878.47it/s]
>>> a2 = ytree.load(fn)
```

By default, all trees and all fields will be saved, but this can be customized with the `trees` and `fields` keywords.

For convenience, individual trees can also be saved by calling *save_tree*.

```
>>> fn = a[0].save_tree()
Creating field arrays [1/1]: 100%|| 4897/4897 [00:00<00:00, 13711286.17it/s]
>>> a2 = ytree.load(fn)
```

1.5 An Important Note on Comoving and Proper Units

Users of `yt` are likely familiar with conversion from proper to comoving reference frames by adding "cm" to a unit. For example, proper "Mpc" becomes comoving with "Mpccm". This conversion relies on all the data being associated with a single redshift. This is not possible here because the dataset has values for multiple redshifts. To account for this, the proper and comoving unit systems are set to be equal to each other.

```
>>> print (a.box_size)
100.0 Mpc/h
>>> print (a.box_size.to("Mpccm/h"))
100.0 Mpccm/h
```

Data should be assumed to be in the reference frame in which it was saved. For length scales, this is typically the comoving frame. When in doubt, the safest unit to use for lengths is "unitary", which a system normalized to the box size.

```
>>> print (a.box_size.to("unitary"))
1.0 unitary
```

1.6 Fields in ytree

ytree supports multiple types of fields, each representing numerical values associated with each halo in the `Arbor`. These include the *native fields* stored on disk, *alias fields*, *derived fields*, and *analysis fields*.

1.6.1 The Field Info Container

Each *`Arbor`* contains a dictionary, called `field_info`, with relevant information for each available field. This information can include the units, type of field, any dependencies or aliases, and things relevant to reading the data from disk.

```
>>> import ytree
>>> a = ytree.load("tree_0_0_0.dat")
>>> print (a.field_info["Rvir"])
{'description': 'Halo radius (kpc/h comoving).', 'units': 'kpc/h ', 'column': 11,
 'aliases': ['virial_radius']}
>>> print (a.field_info["mass"])
{'type': 'alias', 'units': 'Msun', 'dependencies': ['Mvir']}
```

1.6.2 Fields on Disk

Every field stored in the dataset's files should be available within the `Arbor`. The `field_list` contains a list of all fields on disk with their native names.

```
>>> print (a.field_list)
['scale', 'id', 'desc_scale', 'desc_id', 'num_prog', ...]
```

1.6.3 Alias Fields

Because the various dataset formats use different naming conventions for similar fields, ytree allows fields to be referred to by aliases. This allows for a universal set of names for the most common fields. Many are added by default, including "mass", "virial_radius", "position_<xyz>", and "velocity_<xyz>". The list of available alias and derived fields can be found in the `derived_field_list`.

```
print (a.derived_field_list)
['uid', 'desc_uid', 'scale_factor', 'mass', 'virial_mass', ...]
```

Additional aliases can be added with `add_alias_field`.

```
>>> a.add_alias_field("amount_of_stuff", "mass", units="kg")
>>> print (a["amount_of_stuff"])
[  1.30720461e+45,   1.05085632e+45,   1.03025691e+45, ...
1.72691772e+42,   1.72691772e+42,   1.72691772e+42]) kg
```

1.6.4 Derived Fields

Derived fields are functions of existing fields, including other derived and alias fields. New derived fields are created by providing a defining function and calling `add_derived_field`.

```
>>> def potential_field(data):
...     # data.arbor points to the parent Arbor
...     return data["mass"] / data["virial_radius"]
...
>>> a.add_derived_field("potential", potential_field, units="Msun/Mpc")
[  2.88624262e+14   2.49542426e+14   2.46280488e+14, ...
3.47503685e+12   3.47503685e+12   3.47503685e+12] Msun/Mpc
```

Field functions should only take a single argument, representing the entity for which the field is defined. This argument will also have access to the parent `Arbor`.

1.6.5 Analysis Fields

Analysis fields provide a means for saving the results of complicated analysis for any halo in the `Arbor`. This would be operations beyond derived fields, for example, things that might require loading the original simulation snapshots. New analysis fields are created with `add_analysis_field` and are initialized to zero.

```
>>> a.add_analysis_field("saucer_sections", units="m**2")
>>> print (a[0]["tree", "saucer_sections"])
[ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
  0.,  0.,] m**2
>>> import numpy as np
>>> for t in a[0]["tree"]:
...     t["saucer_sections"] = np.random.random() # complicated analysis
...
>>> print (a[0]["tree", "saucer_sections"])
[ 0.33919263  0.79557815  0.38264336  0.53073945  0.09634924  0.6035886, ...
  0.9506636   0.9094426   0.85436984  0.66779632  0.58816873] m**2
```

Analysis fields will be automatically saved when the `Arbor` is saved with *`save_arbor`*.

1.7 Making Merger-trees from Gadget FoF/Subfind

The ytree `TreeFarm` can compute merger-trees either for all halos, starting at the beginning of the simulation, or for specific halos, starting at the final output and moving backward. These two use-cases are covered separately. Halo catalogs must be in the form created by the Gadget FoF halo finder or Subfind substructure finder.

1.7.1 Computing a Full Merger-tree

`TreeFarm` accepts a `yt` time-series object over which the merger-tree will be computed.

```
import yt
import ytree

ts = yt.DatasetSeries("data/groups_*/*.0.hdf5")
my_tree = ytree.TreeFarm(ts)
my_tree.trace_descendents("Group", filename="all_halos/")
```

The first argument to `trace_descendents` specifies the type of halo object to use. This will typically be either "Group" for FoF groups or Subhalo for Subfind groups. This process will create a new halo catalogs with the additional field representing the descendent ID for each halo. These can be loaded using `yt` like any other catalogs. Once complete, the final merger-tree can be *loaded into ytree*.

1.7.2 Computing a Targeted Merger-tree

Computing a full merger-tree can be extremely expensive when the simulation is large. Instead, merger-trees can be created for specific halos in the final dataset, then working backward. Below is an example of computing the merger-tree for only the most massive halo.

```
import yt
import ytree

ds = yt.load("fof_subfind/groups_025/fof_subhalo_tab_025.0.hdf5")
i_max = np.argmax(ds.r["Group", "particle_mass"])
my_id = ds.r["particle_identifier"][i_max]

ts = yt.DatasetSeries("data/groups_*/*.0.hdf5")
my_tree = ytree.TreeFarm(ts)
my_tree.trace_ancestors("Group", my_id, filename="my_halo/")
```

Just as above, the resulting catalogs can then be loaded into a *TreeFarm Arbor*.

1.7.3 Optimizing Merger-tree Creation

Computing merger-trees can often be an expensive task. Below are some tips for speeding up the process.

Running in Parallel

ytree uses the parallel capabilities of `yt` to divide up the halo ancestor/descendent search over multiple processors. In order to do this, `yt` must be set up to run in parallel. See here for instructions. Once this is done, a call to `yt.enable_parallelism()` must be added to your script.

```
import yt
yt.enable_parallelism()
import ytree

ts = yt.DatasetSeries("data/groups_*/*.0.hdf5")
my_tree = ytree.TreeFarm(ts)
my_tree.trace_descendents("Group", filename="all_halos/")
```

That script must then be run with mpirun.

```
mpirun -np 4 python my_script.py
```

Optimizing Halo Candidate Selection

Halo ancestors and descendents are typically found by comparing particle IDs between two halos. The method of selecting which halos should be compared can greatly affect performance. By default, `TreeFarm` will compare a halo against all halos in the next dataset. This is both the most robust and slowest method of matching ancestors and descendents. A smarter method is to select candidate matches from only a region around the target halo. For example, `TreeFarm` can be configured to select halos from a sphere centered on the current halo.

```
my_tree = ytree.TreeFarm(ts)
my_tree.set_selector("sphere", "virial_radius", factor=5)
my_tree.trace_descendents("Group", filename="all_halos/")
```

In the above example, candidate halos will be selected from a sphere that is five times the value of the "virial_radius" field. While this will speed up the calculation, a match will not be found if the ancestor/descendent is outside of this region. Some experimentation is recommended to find the optimal balance between speed and robustness.

Currently, the "sphere" selector is the only other selection method implemented, although others can be created easily. For an example, see `sphere_selector`.

Searching for Fewer Ancestors

When computing a merger-tree for specific halos (*Computing a Targeted Merger-tree*), you only be interested in the most massive or the few most massive progenitors. If this is the case, `TreeFarm` can be configured to end the ancestor search when these have been found, rather than searching for all possible progenitors.

The `set_ancestry_filter` function places a filter on which ancestors of any given halo will be returned and followed in successive rounds of the merger-tree process. The "most_massive" filter instructs the `TreeFarm` to only keep the most massive ancestor. This will greatly reduce the number of halos included in the merger-tree and, therefore, speed up the calculation considerably. For an example of how to create a new filter, see `most_massive`.

The filtering will only occur after all candidates have been checked for ancestry. An additional operation an be added to end the ancestry search after certain criteria have been met. In the call to `set_ancestry_short` below, the ancestry search will end as soon as an ancestor with at least 50% of the mass of the target halo has been found. For an example of how to create a new function of this type, see `most_massive`.

```
ts = yt.DatasetSeries("data/groups_*/*.0.hdf5")
my_tree = ytree.TreeFarm(ts)
my_tree.trace_ancestors("Group", my_id, filename="my_halo/")
my_tree.set_ancestry_filter("most_massive")
my_tree.set_ancestry_short("above_mass_fraction", 0.5)
```

1.8 Community Code of Conduct

ytree is a project by members of the yt community. Anyone who wishes to participate in the ytree community is expected to honor the yt Community Code of Conduct.

1.9 Contributing to ytree

ytree is a community project and it will be better with your contribution.

Contributions are welcome in the form of code, documentation, or just about anything. If you're interested in getting involved, please do!

ytree is developed using the same conventions as yt. The yt Developer Guide is a good reference for code style, communication with other developers, working with git, and issuing pull requests. For information specific to ytree, such as testing and adding support for new file formats, see the ytree Developer Guide.

If you'd like some help, you can contact the yt developers list. We're there and ready to help!

1.10 Developer Guide

ytree is developed using the same conventions as yt. The yt Developer Guide is a good reference for code style, communication with other developers, working with git, and issuing pull requests. Below is a brief guide of aspects that are specific to ytree.

1.10.1 Contributing in a Nutshell

Step zero, get out of that nutshell!

After that, the process for making contributions to ytree is roughly as follows:

1. Fork the main ytree repository.
2. Create a new branch.
3. Make changes.
4. Run tests. Return to step 3, if needed.
5. Issue pull request.

The yt Developer Guide and github documentation will help with the mechanics of git and pull requests.

1.10.2 Testing

The ytree source comes with a series of tests that can be run to ensure nothing unexpected happens after changes have been made. These tests will automatically run when a pull request is issued or updated, but they can also be run locally very easily. At present, the suite of tests for ytree takes about three minutes to run.

Testing Data

The first order of business is to obtain the sample datasets. See *Sample Data* for how to do so. Next, ytree must be configure to know the location of this data. This is done by creating a configuration file in your home directory at the location `~/.config/ytree/ytreerc`.

```
$ mkdir -p ~/.config/ytree
$ echo [ytree] > ~/.config/ytree/ytreerc
$ echo test_data_dir = /Users/britton/ytree_data >> ~/.config/ytree/ytreerc
$ cat ~/.config/ytree/ytreerc
[ytree]
test_data_dir = /Users/britton/ytree_data
```

This path should point to the outer directory containing all the sample datasets.

Run the Tests

Before running the tests, you will the `pytest` and `flake8` packages. These can be installed with pip.

```
$ pip install pytest flake8
```

Once installed, the tests are run from the top level of the ytree source.

```
$ pytest tests
============================= test session starts ==============================
platform darwin -- Python 3.6.0, pytest-3.0.7, py-1.4.32, pluggy-0.4.0
rootdir: /Users/britton/Documents/work/yt/extensions/ytree/ytree, inifile:
collected 16 items

tests/test_arbors.py ........
tests/test_flake8.py .
tests/test_saving.py ...
tests/test_treefarm.py ..
tests/test_ytree_1x.py ..

========================== 16 passed in 185.03 seconds ==========================
```

1.10.3 Adding Support for a New Format

The *Arbor* class is reasonably generalized such that adding support for a new file format should be relatively straightforward. The existing frontends also provide guidance for what must be done. Below is a brief guide for how to proceed. If you are interested in doing this, we will be more than happy to help!

Where do the files go?

As in yt, the code specific to one file format is referred to as a "frontend". Within the ytree source, each frontend is located in its own directory within `ytree/arbor/frontends`. Name your directory using lowercase and underscores and put it in there.

To allow your frontend to be directly importable at run-time, add the name to the `_frontends` list in `ytree/arbor/frontends/api.py`.

Building Your Frontend

To build a new frontend, you will need to make frontend-specific subclasses for a few components. The easiest way to do this is to start with a blank `Arbor` subclass first. Create a sample script that loads your data with `load`, prints the number of trees, and queries some fields. Within the base classes, the necessary functions will raise a `NotImplementedError` if you have not added them yet. Keep running your script and implementing the function raising this error and before you know it, you'll be done.

The components and the files in which they belong are:

1. The `Arbor` itself (`arbor.py`).
2. The file i/o (`io.py`).
3. Recognizing frontend-specific fields (`fields.py`).

In addition to this, you will need to add a file called `__init__.py`, which will allow your code to be imported. This file should minimally import the frontend-specific `Arbor` class. For example, the consistent-trees `__init__.py` looks like this:

```
from ytree.arbor.frontends.consistent_trees.arbor import \
    ConsistentTreesArbor
```

Two Types of Arbors

There are generally two types of merger-tree data that ytree ingests:

1. all merger-tree data (full trees, halos, etc.) contained within a single file. An example of this is the consistent-trees frontend.

2. halos in files grouped by redshift (halo catalogs) that contain the halo id for the descendent halo which lives in the next catalog. An example of this is the rockstar frontend.

Depending on your case, different base classes should be subclassed. This is discussed below.

The `_is_valid` Function

Within every `Arbor` subclass should appear a function called `_is_valid`. This function is used by `load` to determine if the provide file is the correct type. This function can examine the file's naming convention and/or open it and inspect its contents, whatever is required to uniquely identify your frontend. Have a look at the various examples.

Merger-Tree Data in One File (or a few)

If this is your case, then the consistent-trees and "ytree" frontends are the best examples to follow.

In `arbor.py`, your subclass of `Arbor` should implement two functions, `_parse_parameter_file` and `_plant_trees`.

`_parse_parameter_file`: This is the first thing called when your dataset is loaded. It is responsible for determining things like box size, cosmological parameters, and the list of fields.

`_plant_trees`: This function is responsible for constructing the array containing the roots of all trees in the `Arbor`. This should not fully build the trees, but just create *`TreeNode`* instances for each root and put them in the array.

In `io.py`, you will implement the machinery responsible for reading field data from disk. You must create a subclass of the *`TreeFieldIO`* class and implement the `_read_fields` function. This function accepts a single root node (a `TreeNode` that is the root of a tree) and a list of fields and should return a dictionary with NumPy arrays for each field.

Halo Catalog-style Data

If this is your case, then the rockstar and tree_farm frontends are the best examples to follow.

For this type of data, you will subclass the *`CatalogArbor`* class, which is itself a subclass of *`Arbor`* designed for this type of data.

In `arbor.py`, your subclass should implement two functions, `_parse_parameter_file` and `_get_data_files`. The purpose of `_parse_parameter_file` is described above.

`_get_data_files`: This type of data is usually loaded by providing one of the set of files. This function needs to figure out how many other files there are and their names and construct a list to be saved.

In `io.py`, you will create a subclass of *`CatalogDataFile`* and implement two functions: `_parse_header` and `_read_fields`.

`_parse_header`: This function reads any metadata specific to this halo catalog. For exmaple, you might get the current redshift here.

`_read_fields`: This function is responsible for reading field data from disk. This should minimally take a list of fields and return a dictionary with NumPy arrays for each field for all halos contained in the file. It should also, optionally, take a list of *`TreeNode`* instances and return fields only for them.

Universal Field Aliases (`fields.py`)

By creating aliases to standardized names, scripts can be run on multiple types of data with little or no alteration for frontend-specific field names. In `fields.py`, you will subclass the *`FieldInfoContainer`* class. Most likely, all that will need to be done is to add the list of fields with universal alias names. This is done with the `alias_fields` tuple. The convention for each entry is (alias name, name on disk, field units).

```
from ytree.arbor.fields import \
     FieldInfoContainer

class NewCodeFieldInfo(FieldInfoContainer):
    alias_fields = (
        ("mass", "Mass", "Msun"),
        ("position_x", "PX", "Mpc/h"),
        ...
    )
```

You made it!

That's all there is to it! Now you too can do whatever it is people do with merger-trees. There are probably important things that were left out of this document. If you find any, please consider making an addition or opening an issue. If you're stuck anywhere, don't hesitate to ask for help. If you've gotten this far, we really want to see you make it to the finish!

Everyone Loves Samples

It would be especially great if you could provide a small sample dataset with your new frontend, something less than a few hundred MB if possible. This will ensure that your new frontend never gets broken and will also help new users get started. Once you have some data, make an addition to the arbor tests by following the example in `tests/test_arbors.py`. Then, contact Britton Smith to arrange for your sample data to be added to the ytree data collection on the yt Hub.

Ok, now you're totally done. Take the rest of the afternoon off.

1.11 Help

If you encounter problems, we want to help and there are lots of places to get help. As an extension of the yt project, we are members of the yt community. Any questions regarding ytree can be posted to the yt users list. You will also find interactive help on the yt slack channel.

Bugs and feature requests can also be posted on the ytree issues page.

See you out there!

1.12 API Reference

1.12.1 Working with Merger-Trees

The *`load`* can load all supported merger-tree formats. Once loaded, the *`save_arbor`* and *`save_tree`* functions can be used to save the entire arbor or individual trees.

load(filename[, method])	Load an Arbor, determine the type automatically.
Arbor(filename)	Base class for all Arbor classes.
save_arbor([filename, fields, trees, ...])	Save the arbor to a file.
save_tree([filename, fields])	Save the tree to a file.
set_selector(selector, *args, **kwargs)	Sets the tree node selector to be used.
TreeNodeSelector(function[, args, kwargs])	The TreeNodeSelector is responsible for choosing which one of a halo's ancestors to return when querying the line of main progenitors for a halo.
add_tree_node_selector(name, function)	Add a TreeNodeSelector to the registry of known selectors, so they can be chosen with *set_selector*.
max_field_value(ancestors, field)	Return the TreeNode with the maximum value of the given field.
min_field_value(ancestors, field)	Return the TreeNode with the minimum value of the given field.

ytree.arbor.arbor.load

ytree.arbor.arbor.**load**(*filename*, *method=None*)

Load an Arbor, determine the type automatically.

filename [string] Input filename.

method [optional, string] The type of Arbor to be loaded. Existing types are: ConsistentTrees, Rockstar, TreeFarm, YTree. If not given, the type will be determined based on characteristics of the input file.

Arbor

```
>>> import ytree
>>> # saved Arbor
>>> a = ytree.load("arbor/arbor.h5")
>>> # consistent-trees output
>>> a = ytree.load("rockstar_halos/trees/tree_0_0_0.dat")
>>> # Rockstar catalogs
>>> a = ytree.load("rockstar_halos/out_0.list")
>>> # TreeFarm catalogs
>>> a = ytree.load("my_halos/fof_subhalo_tab_025.0.h5")
```

ytree.arbor.arbor.Arbor

class ytree.arbor.arbor.**Arbor**(*filename*)

Base class for all Arbor classes.

Loads a merger-tree output file or a series of halo catalogs and create trees, stored in an array in `trees`. Arbors can be saved in a universal format with *save_arbor*. Also, provide some convenience functions for creating YTArrays and YTQuantities and a cosmology calculator.

__init__(*filename*)

Initialize an Arbor given an input file.

Methods

__init__(filename)	Initialize an Arbor given an input file.
add_alias_field(alias, field[, units, force_add])	Add a field as an alias to another field.
add_analysis_field(name, units)	Add an empty field to be filled by analysis operations.
add_derived_field(name, function[, units, ...])	Add a field that is a function of other fields.
is_grown(tree_node)	Return True if a tree has been fully assembled, i.e., the hierarchy of ancestor tree nodes has been built.
is_setup(tree_node)	Return True if arrays of uids and descendent uids have been read in.
query(key)	If given a string, return an array of field values for the roots of all trees.
save_arbor([filename, fields, trees, ...])	Save the arbor to a file.
select_halos(criteria[, trees, select_from, ...])	Select halos from the arbor based on a set of criteria given as a string.
set_selector(selector, *args, **kwargs)	Sets the tree node selector to be used.

ytree.arbor.arbor.Arbor.save_arbor

Arbor.**save_arbor**(*filename='arbor'*, *fields=None*, *trees=None*, *max_file_size=524288*)

Save the arbor to a file.

The saved arbor can be re-loaded as an arbor.

filename [optional, string] Output file keyword. If filename ends in ".h5", the main header file will be just that. If not, filename will be <filename>/<basename>.h5. Default: "arbor".

fields [optional, list of strings] The fields to be saved. If not given, all fields will be saved.

header_filename [string] The filename of the saved arbor.

```
>>> import ytree
>>> a = ytree.load("rockstar_halos/trees/tree_0_0_0.dat")
>>> fn = a.save_arbor()
>>> # reload it
>>> a2 = ytree.load(fn)
```

ytree.arbor.tree_node.TreeNode.save_tree

TreeNode.**save_tree**(*filename=None*, *fields=None*)

Save the tree to a file.

The saved tree can be re-loaded as an arbor.

filename [optional, string] Output file keyword. Main header file will be named <filename>/<filename>.h5. Default: "tree_<uid>".

fields [optional, list of strings] The fields to be saved. If not given, all fields will be saved.

filename [string] The filename of the saved arbor.

```
>>> import ytree
>>> a = ytree.load("rockstar_halos/trees/tree_0_0_0.dat")
>>> # save the first tree
>>> fn = a[0].save_tree()
>>> # reload it
>>> a2 = ytree.load(fn)
```

ytree.arbor.arbor.Arbor.set_selector

Arbor.**set_selector**(*selector*, **args*, ***kwargs*)

Sets the tree node selector to be used.

This sets the manner in which halo progenitors are chosen from a list of ancestors. The most obvious example is to select the most massive ancestor.

selector [string] Name of the selector to be used.

Any additional arguments and keywords to be provided to the selector function should follow.

```
>>> import ytree
>>> a = ytree.load("rockstar_halos/trees/tree_0_0_0.dat")
>>> a.set_selector("max_field_value", "mass")
```

ytree.arbor.tree_node_selector.TreeNodeSelector

class ytree.arbor.tree_node_selector.**TreeNodeSelector**(*function*, *args=None*, *kwargs=None*)

The TreeNodeSelector is responsible for choosing which one of a halo's ancestors to return when querying the line of main progenitors for a halo.

ancestors [list of TreeNode objects] List of TreeNode objects from which to select.

The function should return a single TreeNode.

```
>>> import ytree
>>> def max_value(ancestors, field):
...     vals = np.array([a[field] for a in ancestors])
...     return ancestors[np.argmax(vals)]
>>> ytree.add_tree_node_selector("max_field_value", max_value)
>>> a = ytree.load("tree_0_0_0.dat")
>>> a.set_selector("max_field_value", "mass")
>>> print (a[0]["prog"])
```

__init__(*function*, *args=None*, *kwargs=None*)

Methods

__init__(function[, args, kwargs])

ytree.arbor.tree_node_selector.add_tree_node_selector

ytree.arbor.tree_node_selector.**add_tree_node_selector**(*name*, *function*)

Add a TreeNodeSelector to the registry of known selectors, so they can be chosen with *set_selector*.

name [string] Name of the selector.

function [callable] The associated function.

```
>>> import ytree
>>> def max_value(ancestors, field):
```

```
...     vals = np.array([a[field] for a in ancestors])
...     return ancestors[np.argmax(vals)]
>>> ytree.add_tree_node_selector("max_field_value", max_value)
>>> a = ytree.load("tree_0_0_0.dat")
>>> a.set_selector("max_field_value", "mass")
>>> print (a[0]["prog"])
```

ytree.arbor.tree_node_selector.max_field_value

ytree.arbor.tree_node_selector.**max_field_value**(*ancestors*, *field*)

Return the TreeNode with the maximum value of the given field.

ancestors [list of TreeNode objects] List of TreeNode objects from which to select.

field [string] Field to be used for selection.

TreeNode object

ytree.arbor.tree_node_selector.min_field_value

ytree.arbor.tree_node_selector.**min_field_value**(*ancestors*, *field*)

Return the TreeNode with the minimum value of the given field.

ancestors [list of TreeNode objects] List of TreeNode objects from which to select.

field [string] Field to be used for selection.

TreeNode object

1.12.2 Making Merger-Trees

TreeFarm(time_series[, setup_function])	TreeFarm is the merger-tree creator for Gadget FoF and Subfind halo catalogs.
trace_ancestors(halo_type, root_ids[, ...])	Trace the ancestry of a given set of halos.
trace_descendents(halo_type[, fields, filename])	Trace the descendents of all halos.
set_selector(selector, *args, **kwargs)	Set the method for selecting candidate halos for tracing halo ancestry.
set_ancestry_checker(ancestry_checker, ...)	Set the method for determing if a halo is the ancestor of another halo.
set_ancestry_filter(ancestry_filter, *args, ...)	Select a method for determining which ancestors are kept.
set_ancestry_short(ancestry_short, *args, ...)	Select a method for cutting short the ancestry search.
AncestryChecker(function[, args, kwargs])	An AncestryCheck is a function that is responsible for determining whether one halo is an ancestor of another.
add_ancestry_checker(name, function)	Add an ancestry checking function to the registry.
common_ids(descendent_ids, ancestor_ids[, ...])	Determine if at least a given fraction of ancestor's member particles are in the descendent.
AncestryFilter(function[, args, kwargs])	An AncestryFilter takes a halo and a list of ancestors and returns a filtered list of filtered list of ancestors.
add_ancestry_filter(name, function)	Add an ancestry filter function to the registry.
most_massive(halo, ancestors)	Return only the most massive ancestor.
AncestryShort(function[, args, kwargs])	An AncestryShort takes a halo and an ancestor halo and determines if the ancestry search should come to an end.

Continued on next page

Table 1.4 – continued from previous page

add_ancestry_short(name, function)	Add an ancestry short-out function to the registry.
above_mass_fraction(halo, ancestor, fraction)	Return only the most massive ancestor.
HaloSelector(function[, args, kwargs])	A HaloSelector is a function that is responsible for creating a list of ids of halos that are potentially ancestors of a given halo.
add_halo_selector(name, function)	Add a HaloSelector to the registry of known selectors, so they can be chosen with *set_selector*.
sphere_selector(hc, ds2, radius_field[, ...])	Select halos within a sphere around the target halo.
all_selector(hc, ds2)	Return all halos from the ancestor dataset.

ytree.tree_farm.tree_farm.TreeFarm

class ytree.tree_farm.tree_farm.**TreeFarm**(*time_series*, *setup_function=None*)

TreeFarm is the merger-tree creator for Gadget FoF and Subfind halo catalogs.

TreeFarm can be used to create a merger-tree for the full set of halos, starting from the first catalog, or can be used to trace the ancestry of specific halos, starting from the last catalog. The merger-tree process will create a new set of halo catalogs, containing necessary fields (positions, velocities, masses), user-requested fields, and descendent IDs for each halo. These halo catalogs can be loaded at yt datasets.

time_series [yt DatasetSeries object] A yt time-series object containing the datasets over which the merger-tree will be calculated.

setup_function [optional, callable] A function that accepts a yt Dataset object and performs any setup, such as adding derived fields.

To create a full merger tree:

```
>>> import nummpy as np
>>> import yt
>>> import ytree
>>> from ytree.tree_farm import TreeFarm
>>> ts = yt.DatasetSeries("data/groups_*/fof_subhalo_tab*.0.hdf5")
>>> my_tree = TreeFarm(ts)
>>> my_tree.trace_descendents("Group", filename="all_halos/")
>>> a = ytree.load("all_halos/fof_subhalo_tab_000.0.h5")
>>> m = a["particle_mass"]
>>> i = np.argmax(m)
>>> print (a.trees[i]["prog", "particle_mass").to("Msun/h"))
```

To create a merger tree for a specific halo or set of halos:

```
>>> import nummpy as np
>>> import yt
>>> import ytree
>>> from ytree.tree_farm import TreeFarm
>>> ts = yt.DatasetSeries("data/groups_*/fof_subhalo_tab*.0.hdf5")
>>> ds = yt[-1]
>>> i = np.argmax(ds.r["Group", "particle_mass"].d)
>>> my_ids = ds.r["Group", "particle_identifier"][i_max]
>>> my_tree = TreeFarm(ts)
>>> my_tree.set_ancestry_filter("most_massive")
>>> my_tree.set_ancestry_short("above_mass_fraction", 0.5)
>>> my_tree.trace_ancestors("Group", my_ids, filename="my_halos/")
>>> a = ytree.load("my_halos/fof_subhalo_tab_025.0.h5")
>>> print (a[0]["prog", "particle_mass").to("Msun/h"))
```

__init__(*time_series*, *setup_function=None*)

Methods

__init__(time_series[, setup_function])	
set_ancestry_checker(ancestry_checker, ...)	Set the method for determing if a halo is the ancestor of another halo.
set_ancestry_filter(ancestry_filter, *args, ...)	Select a method for determining which ancestors are kept.
set_ancestry_short(ancestry_short, *args, ...)	Select a method for cutting short the ancestry search.
set_selector(selector, *args, **kwargs)	Set the method for selecting candidate halos for tracing halo ancestry.
trace_ancestors(halo_type, root_ids[, ...])	Trace the ancestry of a given set of halos.
trace_descendents(halo_type[, fields, filename])	Trace the descendents of all halos.

ytree.tree_farm.tree_farm.TreeFarm.trace_ancestors

TreeFarm.**trace_ancestors**(*halo_type*, *root_ids*, *fields=None*, *filename=None*)

Trace the ancestry of a given set of halos.

A merger-tree for a specific set of halos will be created, starting with the last halo catalog and moving backward.

halo_type [string] The type of halo, typically "FOF" for FoF groups or "Subfind" for subhalos.

root_ids [integer or array of integers] The halo IDs from the last halo catalog for the targeted halos.

fields [optional, list of strings] List of additional fields to be saved to halo catalogs.

filename [optional, string] Directory in which merger-tree catalogs will be saved.

ytree.tree_farm.tree_farm.TreeFarm.trace_descendents

TreeFarm.**trace_descendents**(*halo_type*, *fields=None*, *filename=None*)

Trace the descendents of all halos.

A merger-tree for all halos will be created, starting with the first halo catalog and moving forward.

halo_type [string] The type of halo, typically "FOF" for FoF groups or "Subfind" for subhalos.

fields [optional, list of strings] List of additional fields to be saved to halo catalogs.

filename [optional, string] Directory in which merger-tree catalogs will be saved.

ytree.tree_farm.tree_farm.TreeFarm.set_selector

TreeFarm.**set_selector**(*selector*, **args*, ***kwargs*)

Set the method for selecting candidate halos for tracing halo ancestry.

The default selector is "all", i.e., check every halo for a possible match. This can be slow. The "sphere" selector can be used to specify that only halos within some sphere be checked.

selector [string] Name of selector.

ytree.tree_farm.tree_farm.TreeFarm.set_ancestry_checker

TreeFarm.**set_ancestry_checker**(*ancestry_checker*, **args*, ***kwargs*)

Set the method for determing if a halo is the ancestor of another halo.

The default method defines an ancestor as a halo where at least 50% of its particles are found in the descendent.

ancestry_checker [string] Name of checking method.

ytree.tree_farm.tree_farm.TreeFarm.set_ancestry_filter

TreeFarm.**set_ancestry_filter**(*ancestry_filter*, **args*, ***kwargs*)

Select a method for determining which ancestors are kept. The kept ancestors will have their ancestries tracked. This can be used to speed up merger-trees for targeted halos by specifying that only the most massive ancestor be kept.

ancestry_filter [string] Name of filter method.

ytree.tree_farm.tree_farm.TreeFarm.set_ancestry_short

TreeFarm.**set_ancestry_short**(*ancestry_short*, **args*, ***kwargs*)

Select a method for cutting short the ancestry search.

This can be used to speed up merger-trees for targeted halos by specifying that the search come to an end when an ancestor with greater than 50% of the halo's mass has been found, thereby ensuring that the most massive halo has already been found.

ancestry_short [string] Name of short-out method.

ytree.tree_farm.ancestry_checker.AncestryChecker

class ytree.tree_farm.ancestry_checker.**AncestryChecker**(*function*, *args=None*, *kwargs=None*)

An AncestryCheck is a function that is responsible for determining whether one halo is an ancestor of another.

descendent_ids [list of ints] Member ids for first halo.

ancestor_ids [list of int] Member ids for second halo.

The function should return True or False.

__init__(*function*, *args=None*, *kwargs=None*)

Methods

__init__(function[, args, kwargs])

ytree.tree_farm.ancestry_checker.add_ancestry_checker

ytree.tree_farm.ancestry_checker.**add_ancestry_checker**(*name*, *function*)

Add an ancestry checking function to the registry.

ytree.tree_farm.ancestry_checker.common_ids

ytree.tree_farm.ancestry_checker.**common_ids**(*descendent_ids*, *ancestor_ids*, *threshold=0.5*)

Determine if at least a given fraction of ancestor's member particles are in the descendent.

descendent_ids [list of ints] Member ids for first halo.

ancestor_ids [list of int] Member ids for second halo.

threshold [float, optional] Critical fraction of ancestor's particles ending up in the descendent to be considered a true ancestor. Default: 0.5.

True or False

ytree.tree_farm.ancestry_filter.AncestryFilter

class ytree.tree_farm.ancestry_filter.**AncestryFilter**(*function*, *args=None*, *kwargs=None*)

An AncestryFilter takes a halo and a list of ancestors and returns a filtered list of filtered list of ancestors. For example, a filter could return only the most massive ancestor.

halo: halo data container Data container of the descendent halo.

ancestors [list of halo data containers] List of data containers for the ancestor halos.

The function should return a list of data containers.

__init__(*function*, *args=None*, *kwargs=None*)

Methods

__init__(function[, args, kwargs])

ytree.tree_farm.ancestry_filter.add_ancestry_filter

ytree.tree_farm.ancestry_filter.**add_ancestry_filter**(*name*, *function*)

Add an ancestry filter function to the registry.

ytree.tree_farm.ancestry_filter.most_massive

ytree.tree_farm.ancestry_filter.**most_massive**(*halo*, *ancestors*)

Return only the most massive ancestor.

halo: halo data container Data container of the descendent halo.

ancestors [list of halo data containers] List of data containers for the ancestor halos.

filtered_ancestors [list of halo data containers] List containing data container of most massive halo.

ytree.tree_farm.ancestry_short.AncestryShort

class ytree.tree_farm.ancestry_short.**AncestryShort**(*function*, *args=None*, *kwargs=None*)

An AncestryShort takes a halo and an ancestor halo and determines if the ancestry search should come to an end.

halo: halo data container Data container of the descendent halo.

ancestor [halo data container] Data container for the ancestor halo.

The function should return True or False.

__init__ (*function*, *args=None*, *kwargs=None*)

Methods

__init__(function[, args, kwargs])

ytree.tree_farm.ancestry_short.add_ancestry_short

ytree.tree_farm.ancestry_short.**add_ancestry_short**(*name*, *function*)
Add an ancestry short-out function to the registry.

ytree.tree_farm.ancestry_short.above_mass_fraction

ytree.tree_farm.ancestry_short.**above_mass_fraction**(*halo*, *ancestor*, *fraction*)
Return only the most massive ancestor.

halo: halo data container Data container of the descendent halo.

ancestor [halo data container] Data containers for the ancestor halo.

True or False

ytree.tree_farm.halo_selector.HaloSelector

class ytree.tree_farm.halo_selector.**HaloSelector**(*function*, *args=None*, *kwargs=None*)
A HaloSelector is a function that is responsible for creating a list of ids of halos that are potentially ancestors of a given halo.

hc [halo container object] Halo container associated with the target halo.

ds2 [halo catalog-type dataset] The dataset of the ancestor halos.

The function should return a list of integers representing the ids of potential halos to check for ancestry.

__init__ (*function*, *args=None*, *kwargs=None*)

Methods

__init__(function[, args, kwargs])

ytree.tree_farm.halo_selector.add_halo_selector

ytree.tree_farm.halo_selector.**add_halo_selector**(*name*, *function*)
Add a HaloSelector to the registry of known selectors, so they can be chosen with *set_selector*.

name [string] Name of the selector.

function [callable] The associated function.

ytree.tree_farm.halo_selector.sphere_selector

ytree.tree_farm.halo_selector.**sphere_selector**(*hc*, *ds2*, *radius_field*, *factor=1*, *min_radius=None*)

Select halos within a sphere around the target halo.

hc [halo container object] Halo container associated with the target halo.

ds2 [halo catalog-type dataset] The dataset of the ancestor halos.

radius_field [str] Name of the field to be used to get the halo radius.

factor [float, optional] Multiplicative factor of the halo radius in which potential halos will be gathered. Default: 1.

min_radius [YTQuantity or tuple of (value, unit)] An absolute minimum radius for the sphere.

my_ids [list of ints] List of ids of potential halos.

ytree.tree_farm.halo_selector.all_selector

ytree.tree_farm.halo_selector.**all_selector**(*hc*, *ds2*)

Return all halos from the ancestor dataset.

hc [halo container object] Halo container associated with the target halo.

ds2 [halo catalog-type dataset] The dataset of the ancestor halos.

my_ids [list of ints] List of ids of potential halos.

1.12.3 Internal Classes

Arbor(filename)	Base class for all Arbor classes.
CatalogArbor(filename)	Base class for Arbors created from a series of halo catalog files where the descendent ID for each halo has been predetermined.
FieldInfoContainer(arbor)	A container for information about fields.
FieldContainer(arbor)	A container for field data.
FakeFieldContainer(arbor[, name])	A fake field data container used to calculate dependencies.
FieldIO(arbor)	Base class for FieldIO classes.
TreeFieldIO(arbor)	IO class for getting fields for a tree.
RootFieldIO(arbor)	IO class for getting fields for the roots of all trees.
FallbackRootFieldIO(arbor)	Class for getting root fields from arbors that have no specialized storage for root fields.
CatalogDataFile(filename, arbor)	Base class for halo catalog files.
TreeNode(uid[, arbor, root])	Class for objects stored in Arbors.
TreeNodeSelector(function[, args, kwargs])	The TreeNodeSelector is responsible for choosing which one of a halo's ancestors to return when querying the line of main progenitors for a halo.
ArborArbor(filename)	Class for Arbors created with ytree version 1.1.0 or earlier.
ArborArborFieldInfo(arbor)	
ArborArborTreeFieldIO(arbor)	
ArborArborRootFieldIO(arbor)	
ConsistentTreesArbor(filename)	Arbors from consistent-trees output files.

Continued on next page

Table 1.10 – continued from previous page

ConsistentTreesFieldInfo(arbor)	
ConsistentTreesTreeFieldIO(arbor)	
RockstarArbor(filename)	Class for Arbors created from Rockstar out_*.list files.
RockstarFieldInfo(arbor)	
RockstarDataFile(filename, arbor)	
TreeFarmArbor(filename)	Class for Arbors created with `TreeFarm`.
TreeFarmFieldInfo(arbor)	
TreeFarmDataFile(filename, arbor)	
TreeFarmTreeFieldIO(arbor)	
YTreeArbor(filename)	Class for Arbors created from the *save_arbor* or *save_tree* functions.
YTreeTreeFieldIO(arbor)	
YTreeRootFieldIO(arbor)	

ytree.arbor.arbor.CatalogArbor

class ytree.arbor.arbor.**CatalogArbor**(*filename*)

Base class for Arbors created from a series of halo catalog files where the descendent ID for each halo has been pre-determined.

__init__(*filename*)

Methods

__init__(filename)	
add_alias_field(alias, field[, units, force_add])	Add a field as an alias to another field.
add_analysis_field(name, units)	Add an empty field to be filled by analysis operations.
add_derived_field(name, function[, units, ...])	Add a field that is a function of other fields.
is_grown(tree_node)	
is_setup(tree_node)	Return True if arrays of uids and descendent uids have been read in.
query(key)	If given a string, return an array of field values for the roots of all trees.
save_arbor([filename, fields, trees, ...])	Save the arbor to a file.
select_halos(criteria[, trees, select_from, ...])	Select halos from the arbor based on a set of criteria given as a string.
set_selector(selector, *args, **kwargs)	Sets the tree node selector to be used.

ytree.arbor.fields.FieldInfoContainer

class ytree.arbor.fields.**FieldInfoContainer**(*arbor*)

A container for information about fields.

__init__(*arbor*)

Methods

__init__(arbor)	

Continued on next page

Table 1.12 – continued from previous page

clear(() -> None. Remove all items from D.)	
copy(() -> a shallow copy of D)	
fromkeys	Returns a new dict with keys from iterable and values equal to value.
get((k[,d]) -> D[k] if k in D, ...)	
items(...)	
keys(...)	
pop((k[,d]) -> v, ...)	If key is not found, d is returned if given, otherwise KeyError is raised
popitem(() -> (k, v), ...)	2-tuple; but raise KeyError if D is empty.
resolve_field_dependencies(fields[, fcache])	Divide fields into those to be read and those to generate.
setdefault((k[,d]) -> D.get(k,d), ...)	
setup_aliases()	Add aliases defined in the alias_fields tuple for each frontend.
setup_derived_fields()	Add stock derived fields.
update(([E, ...)	If E is present and has a .keys() method, then does: for k in E: D[k] = E[k]
values(...)	

ytree.arbor.fields.FieldContainer

class ytree.arbor.fields.**FieldContainer**(*arbor*)

A container for field data.

__init__(*arbor*)

Methods

__init__(arbor)	
clear(() -> None. Remove all items from D.)	
copy(() -> a shallow copy of D)	
fromkeys	Returns a new dict with keys from iterable and values equal to value.
get((k[,d]) -> D[k] if k in D, ...)	
items(...)	
keys(...)	
pop((k[,d]) -> v, ...)	If key is not found, d is returned if given, otherwise KeyError is raised
popitem(() -> (k, v), ...)	2-tuple; but raise KeyError if D is empty.
setdefault((k[,d]) -> D.get(k,d), ...)	
update(([E, ...)	If E is present and has a .keys() method, then does: for k in E: D[k] = E[k]
values(...)	

ytree.arbor.fields.FakeFieldContainer

class ytree.arbor.fields.**FakeFieldContainer**(*arbor*, *name=None*)

A fake field data container used to calculate dependencies.

__init__(*arbor*, *name=None*)

Methods

__init__(arbor[, name])	
clear(() -> None. Remove all items from D.)	
copy(() -> a shallow copy of D.)	
fromkeys	Returns a new dict with keys from iterable and values equal to value.
get((k[,d]) -> D[k] if k in D, ...)	
items(...)	
keys(...)	
pop((k[,d]) -> v, ...)	If key is not found, d is returned if given, otherwise KeyError is raised
popitem(() -> (k, v), ...)	2-tuple; but raise KeyError if D is empty.
setdefault((k[,d]) -> D.get(k,d), ...)	
update(([E, ...)	If E is present and has a .keys() method, then does: for k in E: D[k] = E[k]
values(...)	

ytree.arbor.io.FieldIO

class ytree.arbor.io.**FieldIO**(*arbor*)

Base class for FieldIO classes.

This object is resposible for field i/o for an Arbor.

__init__(*arbor*)

Methods

__init__(arbor)	
get_fields(data_object[, fields])	Load field data for a data object into storage structures.

ytree.arbor.io.TreeFieldIO

class ytree.arbor.io.**TreeFieldIO**(*arbor*)

IO class for getting fields for a tree.

__init__(*arbor*)

Methods

__init__(arbor)	
get_fields(data_object[, fields])	Load field data for a data object into storage structures.

ytree.arbor.io.RootFieldIO

class ytree.arbor.io.**RootFieldIO**(*arbor*)

IO class for getting fields for the roots of all trees.

__init__(*arbor*)

Methods

__init__(arbor)	
get_fields(data_object[, fields])	Load field data for a data object into storage structures.

ytree.arbor.io.FallbackRootFieldIO

class ytree.arbor.io.**FallbackRootFieldIO**(*arbor*)

Class for getting root fields from arbors that have no specialized storage for root fields.

__init__(*arbor*)

Methods

__init__(arbor)	
get_fields(data_object[, fields])	

ytree.arbor.io.CatalogDataFile

class ytree.arbor.io.**CatalogDataFile**(*filename*, *arbor*)

Base class for halo catalog files.

__init__(*filename*, *arbor*)

Methods

__init__(filename, arbor)	

ytree.arbor.tree_node.TreeNode

class ytree.arbor.tree_node.**TreeNode**(*uid*, *arbor=None*, *root=False*)

Class for objects stored in Arbors.

Each TreeNode represents a halo in a tree. A TreeNode knows its halo ID, the level in the tree, and its global ID in the Arbor that holds it. It also has a list of its ancestors. Fields can be queried for it, its progenitor list, and the tree beneath.

__init__(*uid*, *arbor=None*, *root=False*)

Initialize a TreeNode with at least its halo catalog ID and its level in the tree.

Methods

__init__(uid[, arbor, root])	Initialize a TreeNode with at least its halo catalog ID and its level in the tree.
add_ancestor(ancestor)	Add another TreeNode to the list of ancestors.
clear_fields()	If a root node, remove tree-related data structures.

Continued on next page

Table 1.20 – continued from previous page

pwalk()	An iterator over all TreeNodes in the progenitor list, starting with this TreeNode.
query(key)	Return field values for this TreeNode, progenitor list, or tree.
save_tree([filename, fields])	Save the tree to a file.
twalk()	An iterator over all TreeNodes in the tree beneath, starting with this TreeNode.

ytree.arbor.frontends.arborarbor.arbor.ArborArbor

class ytree.arbor.frontends.arborarbor.arbor.**ArborArbor**(*filename*)

Class for Arbors created with ytree version 1.1.0 or earlier.

__init__(*filename*)

Initialize an Arbor given an input file.

Methods

__init__(filename)	Initialize an Arbor given an input file.
add_alias_field(alias, field[, units, force_add])	Add a field as an alias to another field.
add_analysis_field(name, units)	Add an empty field to be filled by analysis operations.
add_derived_field(name, function[, units, ...])	Add a field that is a function of other fields.
is_grown(tree_node)	Return True if a tree has been fully assembled, i.e., the hierarchy of ancestor tree nodes has been built.
is_setup(tree_node)	Return True if arrays of uids and descendent uids have been read in.
query(key)	If given a string, return an array of field values for the roots of all trees.
save_arbor([filename, fields, trees, ...])	Save the arbor to a file.
select_halos(criteria[, trees, select_from, ...])	Select halos from the arbor based on a set of criteria given as a string.
set_selector(selector, *args, **kwargs)	Sets the tree node selector to be used.

ytree.arbor.frontends.arborarbor.fields.ArborArborFieldInfo

class ytree.arbor.frontends.arborarbor.fields.**ArborArborFieldInfo**(*arbor*)

__init__(*arbor*)

Methods

__init__(arbor)	
clear(() -> None. Remove all items from D.)	
copy(() -> a shallow copy of D)	
fromkeys	Returns a new dict with keys from iterable and values equal to value.
get((k[,d]) -> D[k] if k in D, ...)	

Continued on next page

Table 1.22 – continued from previous page

items(...)	
keys(...)	
pop((k[,d]) -> v, ...)	If key is not found, d is returned if given, otherwise KeyError is raised
popitem(() -> (k, v), ...)	2-tuple; but raise KeyError if D is empty.
resolve_field_dependencies(fields[, fcache])	Divide fields into those to be read and those to generate.
setdefault((k[,d]) -> D.get(k,d), ...)	
setup_aliases()	Add aliases defined in the alias_fields tuple for each frontend.
setup_derived_fields()	Add stock derived fields.
update(([E, ...)	If E is present and has a .keys() method, then does: for k in E: D[k] = E[k]
values(...)	

ytree.arbor.frontends.arborarbor.io.ArborArborTreeFieldIO

class ytree.arbor.frontends.arborarbor.io.**ArborArborTreeFieldIO**(*arbor*)

__init__(*arbor*)

Methods

__init__(arbor)	
get_fields(data_object[, fields])	Load field data for a data object into storage structures.

ytree.arbor.frontends.arborarbor.io.ArborArborRootFieldIO

class ytree.arbor.frontends.arborarbor.io.**ArborArborRootFieldIO**(*arbor*)

__init__(*arbor*)

Methods

__init__(arbor)	
get_fields(data_object[, fields])	Load field data for a data object into storage structures.

ytree.arbor.frontends.consistent_trees.arbor.ConsistentTreesArbor

class ytree.arbor.frontends.consistent_trees.arbor.**ConsistentTreesArbor**(*filename*)
Arbors from consistent-trees output files.

__init__(*filename*)
Initialize an Arbor given an input file.

Methods

__init__(filename)	Initialize an Arbor given an input file.
add_alias_field(alias, field[, units, force_add])	Add a field as an alias to another field.
add_analysis_field(name, units)	Add an empty field to be filled by analysis operations.
add_derived_field(name, function[, units, ...])	Add a field that is a function of other fields.
is_grown(tree_node)	Return True if a tree has been fully assembled, i.e., the hierarchy of ancestor tree nodes has been built.
is_setup(tree_node)	Return True if arrays of uids and descendent uids have been read in.
query(key)	If given a string, return an array of field values for the roots of all trees.
save_arbor([filename, fields, trees, ...])	Save the arbor to a file.
select_halos(criteria[, trees, select_from, ...])	Select halos from the arbor based on a set of criteria given as a string.
set_selector(selector, *args, **kwargs)	Sets the tree node selector to be used.

ytree.arbor.frontends.consistent_trees.fields.ConsistentTreesFieldInfo

class ytree.arbor.frontends.consistent_trees.fields.**ConsistentTreesFieldInfo**(*arbor*)

__init__(*arbor*)

Methods

__init__(arbor)	
clear(() -> None. Remove all items from D.)	
copy(() -> a shallow copy of D)	
fromkeys	Returns a new dict with keys from iterable and values equal to value.
get((k[,d]) -> D[k] if k in D, ...)	
items(...)	
keys(...)	
pop((k[,d]) -> v, ...)	If key is not found, d is returned if given, otherwise KeyError is raised
popitem(() -> (k, v), ...)	2-tuple; but raise KeyError if D is empty.
resolve_field_dependencies(fields[, fcache])	Divide fields into those to be read and those to generate.
setdefault((k[,d]) -> D.get(k,d), ...)	
setup_aliases()	Add aliases defined in the alias_fields tuple for each frontend.
setup_derived_fields()	Add stock derived fields.
update(([E, ...)	If E is present and has a .keys() method, then does: for k in E: D[k] = E[k]
values(...)	

ytree.arbor.frontends.consistent_trees.io.ConsistentTreesTreeFieldIO

class ytree.arbor.frontends.consistent_trees.io.**ConsistentTreesTreeFieldIO**(*arbor*)

__init__(*arbor*)

Methods

__init__(arbor)	
get_fields(data_object[, fields])	Load field data for a data object into storage structures.

ytree.arbor.frontends.rockstar.arbor.RockstarArbor

class ytree.arbor.frontends.rockstar.arbor.**RockstarArbor**(*filename*)

Class for Arbors created from Rockstar out_*.list files. Use only descendent IDs to determine tree relationship.

__init__(*filename*)

Methods

__init__(filename)	
add_alias_field(alias, field[, units, force_add])	Add a field as an alias to another field.
add_analysis_field(name, units)	Add an empty field to be filled by analysis operations.
add_derived_field(name, function[, units, ...])	Add a field that is a function of other fields.
is_grown(tree_node)	
is_setup(tree_node)	Return True if arrays of uids and descendent uids have been read in.
query(key)	If given a string, return an array of field values for the roots of all trees.
save_arbor([filename, fields, trees, ...])	Save the arbor to a file.
select_halos(criteria[, trees, select_from, ...])	Select halos from the arbor based on a set of criteria given as a string.
set_selector(selector, *args, **kwargs)	Sets the tree node selector to be used.

ytree.arbor.frontends.rockstar.fields.RockstarFieldInfo

class ytree.arbor.frontends.rockstar.fields.**RockstarFieldInfo**(*arbor*)

__init__(*arbor*)

Methods

__init__(arbor)	
clear(() -> None. Remove all items from D.)	
copy(() -> a shallow copy of D)	
fromkeys	Returns a new dict with keys from iterable and values equal to value.
get((k[,d]) -> D[k] if k in D, ...)	
items(...)	
keys(...)	
pop((k[,d]) -> v, ...)	If key is not found, d is returned if given, otherwise KeyError is raised
popitem(() -> (k, v), ...)	2-tuple; but raise KeyError if D is empty.

Continued on next page

Table 1.29 – continued from previous page

resolve_field_dependencies(fields[, fcache])	Divide fields into those to be read and those to generate.
setdefault((k[,d]) -> D.get(k,d), ...)	
setup_aliases()	Add aliases defined in the alias_fields tuple for each frontend.
setup_derived_fields()	Add stock derived fields.
update(([E, ...)	If E is present and has a .keys() method, then does: for k in E: D[k] = E[k]
values(...)	

ytree.arbor.frontends.rockstar.io.RockstarDataFile

class ytree.arbor.frontends.rockstar.io.**RockstarDataFile**(*filename*, *arbor*)

__init__(*filename*, *arbor*)

Methods

__init__(filename, arbor)	

ytree.arbor.frontends.tree_farm.arbor.TreeFarmArbor

class ytree.arbor.frontends.tree_farm.arbor.**TreeFarmArbor**(*filename*)

Class for Arbors created with TreeFarm.

__init__(*filename*)

Methods

__init__(filename)	
add_alias_field(alias, field[, units, force_add])	Add a field as an alias to another field.
add_analysis_field(name, units)	Add an empty field to be filled by analysis operations.
add_derived_field(name, function[, units, ...])	Add a field that is a function of other fields.
is_grown(tree_node)	
is_setup(tree_node)	Return True if arrays of uids and descendent uids have been read in.
query(key)	If given a string, return an array of field values for the roots of all trees.
save_arbor([filename, fields, trees, ...])	Save the arbor to a file.
select_halos(criteria[, trees, select_from, ...])	Select halos from the arbor based on a set of criteria given as a string.
set_selector(selector, *args, **kwargs)	Sets the tree node selector to be used.

ytree.arbor.frontends.tree_farm.fields.TreeFarmFieldInfo

class ytree.arbor.frontends.tree_farm.fields.**TreeFarmFieldInfo**(*arbor*)

__init__(*arbor*)

Methods

__init__(arbor)	
clear(() -> None. Remove all items from D.)	
copy(() -> a shallow copy of D)	
fromkeys	Returns a new dict with keys from iterable and values equal to value.
get((k[,d]) -> D[k] if k in D, ...)	
items(...)	
keys(...)	
pop((k[,d]) -> v, ...)	If key is not found, d is returned if given, otherwise KeyError is raised
popitem(() -> (k, v), ...)	2-tuple; but raise KeyError if D is empty.
resolve_field_dependencies(fields[, fcache])	Divide fields into those to be read and those to generate.
setdefault((k[,d]) -> D.get(k,d), ...)	
setup_aliases()	Add aliases defined in the alias_fields tuple for each frontend.
setup_derived_fields()	Add stock derived fields.
update(([E, ...)	If E is present and has a .keys() method, then does: for k in E: D[k] = E[k]
values(...)	

ytree.arbor.frontends.tree_farm.io.TreeFarmDataFile

class ytree.arbor.frontends.tree_farm.io.**TreeFarmDataFile**(*filename*, *arbor*)

__init__(*filename*, *arbor*)

Methods

__init__(filename, arbor)	

ytree.arbor.frontends.tree_farm.io.TreeFarmTreeFieldIO

class ytree.arbor.frontends.tree_farm.io.**TreeFarmTreeFieldIO**(*arbor*)

__init__(*arbor*)

Methods

__init__(arbor)	
get_fields(data_object[, fields])	Load field data for a data object into storage structures.

ytree.arbor.frontends.ytree.arbor.YTreeArbor

class ytree.arbor.frontends.ytree.arbor.**YTreeArbor**(*filename*)
Class for Arbors created from the *save_arbor* or *save_tree* functions.

__init__ (*filename*)
Initialize an Arbor given an input file.

Methods

__init__(filename)	Initialize an Arbor given an input file.
add_alias_field(alias, field[, units, force_add])	Add a field as an alias to another field.
add_analysis_field(name, units)	Add an empty field to be filled by analysis operations.
add_derived_field(name, function[, units, ...])	Add a field that is a function of other fields.
is_grown(tree_node)	Return True if a tree has been fully assembled, i.e., the hierarchy of ancestor tree nodes has been built.
is_setup(tree_node)	Return True if arrays of uids and descendent uids have been read in.
query(key)	If given a string, return an array of field values for the roots of all trees.
save_arbor([filename, fields, trees, ...])	Save the arbor to a file.
select_halos(criteria[, trees, select_from, ...])	Select halos from the arbor based on a set of criteria given as a string.
set_selector(selector, *args, **kwargs)	Sets the tree node selector to be used.

ytree.arbor.frontends.ytree.io.YTreeTreeFieldIO

class ytree.arbor.frontends.ytree.io.**YTreeTreeFieldIO**(*arbor*)

__init__ (*arbor*)

Methods

__init__(arbor)	
get_fields(data_object[, fields])	Load field data for a data object into storage structures.

ytree.arbor.frontends.ytree.io.YTreeRootFieldIO

class ytree.arbor.frontends.ytree.io.**YTreeRootFieldIO**(*arbor*)

__init__ (*arbor*)

Methods

__init__(arbor)	
get_fields(data_object[, fields])	Load field data for a data object into storage structures.

CHAPTER 2

Citing ytree

If you use ytree in your work, please cite it as "ytree, written by Britton Smith and the yt project" with a footnote pointing to http://ytree.readthedocs.io.

CHAPTER 3

Search

- search

Index

Symbols

A

C

F

H

L

M

R

S

T

Y