ytree Documentation

Release 3.1.1

Britton Smith

Feb 03, 2022

Contents

1 I want to make merger trees! 3

2 Table of Contents 5
- 2.1 Installation . . . 5
- 2.2 What version do I have? . . . 5
- 2.3 Sample Data . . . 5
- 2.4 An Important Note on Comoving and Proper Units . . . 6
- 2.5 Working with Merger Trees . . . 6
- 2.6 Fields in ytree . . . 19
- 2.7 Plotting Merger Trees . . . 21
- 2.8 Parallel Computing with ytree . . . 24
- 2.9 Analyzing Merger Trees . . . 27
- 2.10 Example Applications . . . 31
- 2.11 Community Code of Conduct . . . 33
- 2.12 Contributing to ytree . . . 34
- 2.13 Developer Guide . . . 34
- 2.14 Help . . . 39
- 2.15 Citing ytree . . . 39
- 2.16 Reference . . . 40

3 Citing ytree 103

4 Search 105

Index 107

`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, symbolic units, parallel functionality, and a framework for performing complex analysis. `ytree` is able to load in merger tree from the following formats:

- *Amiga Halo Finder*
- *Consistent-Trees*
- *Consistent-Trees-HDF5*
- *LHaloTree*
- *LHaloTree-HDF5*
- *MORIA*
- *Rockstar Catalogs*
- *TreeFarm*
- *TreeFrog*

See *Loading Data* for instructions specific to each format. All formats can be *resaved with a universal format* that can be *reloaded with ytree*. Individual trees for single halos can also be saved.

CHAPTER 1

I want to make merger trees!

If you have halo catalog data that can be loaded by yt, then you can use the treefarm package to create merger trees. treefarm was once a part of `ytree`, but is now its own thing.

CHAPTER 2

Table of Contents

2.1 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/ytree-project/ytree. This can be installed by doing:

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

2.2 What version do I have?

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

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

2.3 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 (about 979 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." Individual datasets can also be downloaded from this interface. Finally, the entire collection can 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
```

2.4 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
```

2.5 Working with Merger Trees

The *`Arbor`* class is responsible for loading and providing access to merger tree data. In this document, a loaded merger tree dataset is referred to as an **arbor**. `ytree` provides several different ways to navigate, query, and analyze merger trees. It is recommended that you read this entire section to identify the way that is best for what you want to do.

2.5.1 Loading 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. All examples use the freely available *Sample Data*.

Amiga Halo Finder

The Amiga Halo Finder format stores data in a series of files, with one each per snapshot. Parameters are stored in ".parameters" and ".log" files, halo information in ".AHF_halos" files, and descendent/ancestor links are stored in

".AHF_mtree" files. Make sure to keep all of these together. To load, provide the name of the first ".parameter" file.

```
>>> import ytree
>>> a = ytree.load("ahf_halos/snap_N64L16_000.parameter",
...                 hubble_constant=0.7)
```

Note: Four important notes about loading AHF data:

1. The dimensionless Hubble parameter is not provided in AHF outputs. This should be supplied by hand using the `hubble_constant` keyword. The default value is 1.0.
2. If the ".log" file is named in a unconventional way or cannot be found for some reason, its path can be specified with the `log_filename` keyword argument. If no log file exists, values for `omega_matter`, `omega_lambda`, and `box_size` (in units of Mpc/h) can be provided with keyword arguments named thusly.
3. There will be no ".AHF_mtree" file for index 0 as the ".AHF_mtree" files store links between files N-1 and N.
4. `ytree` is able to load data where the graph has been calculated instead of the tree. However, even in this case, only the tree is preserved in `ytree`. See the Amiga Halo Finder Documentation for a discussion of the difference between graphs and trees.

Consistent-Trees

The consistent-trees format consists of a set of files called "locations.dat", "forests.list", at least one file named something like "tree_0_0_0.dat". For large simulations, there may be a number of these "tree_*.dat" files. After running Rockstar and consistent-trees, these will most likely be located in the "rockstar_halos/trees" directory. The full data set can be loaded by providing the path to the *locations.dat* file.

```
>>> import ytree
>>> a = ytree.load("tiny_ctrees/locations.dat")
```

Alternatively, data from a single tree file can be loaded by providing the path to that file.

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

Consistent-Trees hlist Files

While running consistent-trees, a series of files will be created in the "rockstar_halos/hlists" directory with the naming convention, "hlist_<scale-factor>.list". These are the catalogs that will be combined to make the final output files. However, these files contain roughly 30 additional fields that are not included in the final output. Merger trees can be loaded by providing the path to the first of these files.

```
>>> import ytree
>>> a = ytree.load("ctrees_hlists/hlists/hlist_0.12521.list")
```

Note: Note, loading trees with this method will be slower than using the standard consistent-trees output file as `ytree` will have to assemble each tree across multiple files. This method is not recommended unless the additional fields are necessary.

Consistent-Trees-HDF5

Consistent-Trees-HDF5 is a variant of the consistent-trees format built on HDF5. It is used by the Skies & Universe project. This format allows for access by either *forests* or *trees* as per the definitions above. The data can be stored as either a struct of arrays or an array of structs. Both layouts are supported, but `ytree` is currently optimized for the struct of arrays layout. Field access with struct of arrays will be 1 to 2 orders of magnitude faster than with array of structs.

Datasets from this format consist of a series of HDF5 files with the naming convention, *forest.h5*, *forest_0.5*, . . . , *forest_N.h5*. The numbered files contain the actual data while the *forest.h5* file contains virtual datasets that point to the data files. To load all the data, provide the path to the virtual dataset file:

```
>>> import ytree
>>> a = ytree.load("consistent_trees_hdf5/soa/forest.h5")
```

To load a subset of the full dataset, provide a single data file or a list/tuple of files.

```
>>> import ytree
>>> # single file
>>> a = ytree.load("consistent_trees_hdf5/soa/forest_0.h5")
>>> # multiple data files (sample data only has one)
>>> a = ytree.load(["forest_0.h5", "forest_1.h5"])
```

Access by Forest

By default, `ytree` will load consistent-trees-hdf5 datasets to provide access to each tree, such that `a[N]` will return the Nth tree in the dataset and `a[N]["tree"]` will return all halos in that tree. However, by providing the `access="forest"` keyword to *`load`*, data will be loaded according to the forest it belongs to.

```
>>> import ytree
>>> a = ytree.load("consistent_trees_hdf5/soa/forest.h5",
...                access="forest")
```

In this mode, `a[N]` will return the Nth forest and `a[N]["forest"]` will return all halos in that forest. In forest access mode, the "root" of the forest, i.e., the *`TreeNode`* object returned by doing `a[N]` will be the root of one of the trees in that forest. See *Accessing All Nodes in a Forest* for how to locate all individual trees in a forest.

LHaloTree

The LHaloTree format is typically one or more files with a naming convention like "trees_063.0" that contain the trees themselves and a single file with a suffix ".a_list" that contains a list of the scale factors at the time of each simulation snapshot.

Note: The LHaloTree format loads halos by forest. There is no need to provide the `access="forest"` keyword here.

In addition to the LHaloTree files, `ytree` also requires additional information about the simulation from a parameter file (in Gadget format). At minimum, the parameter file should contain the cosmological parameters `HubbleParam`, `Omega0`, `OmegaLambda`, `BoxSize`, `PeriodicBoundariesOn`, and `ComovingIntegrationOn`, and the unit parameters `UnitVelocity_in_cm_per_s`, `UnitLength_in_cm`, and `UnitMass_in_g`. If not specified explicitly (see below), a file with the extension ".param" will be searched for in the directory containing the LHaloTree files.

If all of the required files are in the same directory, an LHaloTree catalog can be loaded from the path to one of the tree files.

```
>>> import ytree
>>> a = ytree.load("lhalotree/trees_063.0")
```

Both the scale factor and parameter files can be specified explicitly through keyword arguments if they do not match the expected pattern or are located in a different directory than the tree files.

```
>>> a = ytree.load("lhalotree/trees_063.0",
...                parameter_file="lhalotree/param.txt",
...                scale_factor_file="lhalotree/a_list.txt")
```

The scale factors and/or parameters themselves can also be passed explicitly from python.

```
>>> import numpy as np
>>> parameters = dict(HubbleParam=0.7, Omega0=0.3, OmegaLambda=0.7,
...     BoxSize=62500, PeriodicBoundariesOn=1, ComovingIntegrationOn=1,
...     UnitVelocity_in_cm_per_s=100000, UnitLength_in_cm=3.08568e21,
...     UnitMass_in_g=1.989e+43)
>>> scale_factors = [ 0.0078125,  0.012346 ,  0.019608 ,  0.032258 ,  0.047811 ,
...      0.051965 ,  0.056419 ,  0.061188 ,  0.066287 ,  0.071732 ,
...      0.07754  ,  0.083725 ,  0.090306 ,  0.097296 ,  0.104713 ,
...      0.112572 ,  0.120887 ,  0.129675 ,  0.13895  ,  0.148724 ,
...      0.159012 ,  0.169824 ,  0.181174 ,  0.19307  ,  0.205521 ,
...      0.218536 ,  0.232121 ,  0.24628  ,  0.261016 ,  0.27633  ,
...      0.292223 ,  0.308691 ,  0.32573  ,  0.343332 ,  0.361489 ,
...      0.380189 ,  0.399419 ,  0.419161 ,  0.439397 ,  0.460105 ,
...      0.481261 ,  0.502839 ,  0.524807 ,  0.547136 ,  0.569789 ,
...      0.59273  ,  0.615919 ,  0.639314 ,  0.66287  ,  0.686541 ,
...      0.710278 ,  0.734031 ,  0.757746 ,  0.781371 ,  0.804849 ,
...      0.828124 ,  0.851138 ,  0.873833 ,  0.896151 ,  0.918031 ,
...      0.939414 ,  0.960243 ,  0.980457 ,  1.        ]
>>> a = ytree.load("lhalotree/trees_063.0",
...                parameters=parameters,
...                scale_factors=scale_factors)
```

LHaloTree-HDF5

This is the same algorithm as *LHaloTree*, except with data saved in HDF5 files instead of unformatted binary. LHaloTree-HDF5 is one of the formats used by the Illustris-TNG project and is described in detail here. Like *LHaloTree*, this format supports *accessing trees by forest*. The LHaloTree-HDF5 format stores trees in multiple HDF5 files contained within a single directory. Each tree is fully contained within a single file, so loading is possible even when only a subset of all files is present. To load, provide the path to one file.

```
>>> import ytree
>>> a = ytree.load("TNG50-4-Dark/trees_sf1_099.0.hdf5")
```

The files do not contain information on the box size and cosmological parameters of the simulation, but they can be provided by hand, with the box size assumed to be in units of comoving Mpc/h.

```
>>> import ytree
>>> a = ytree.load("TNG50-4-Dark/trees_sf1_099.0.hdf5",
...                box_size=35, hubble_constant=0.6774,
...                omega_matter=0.3089, omega_lambda=0.6911)
```

The LHaloTree-HDF5 format contains multiple definitions of halo mass (see here), and as such, the field alias "mass" is not defined by default. However, the *alias can be created* if one is preferable. This is also necessary to facilitate *Accessing the Progenitor Lineage of a Tree*.

```
>>> a.add_alias_field("mass", "Group_M_TopHat200", units="Msun")
```

MORIA

MORIA is a merger tree extension of the SPARTA code (Diemer 2017; Diemer 2020a). An output from MORIA is a single HDF5 file, whose path should be provided for loading.

```
>>> import ytree
>>> a = ytree.load("moria/moria_tree_testsim050.hdf5")
```

Merger trees in MORIA are organized by *forest*, so printing `a.size` (following the example above) will give the number of forests, not the number of trees. MORIA outputs contain multiple definitions of halo mass (see here), and as such, the field alias "mass" is not defined by default. However, the *alias can be created* if one is preferable. This is also necessary to facilitate *Accessing the Progenitor Lineage of a Tree*.

```
>>> a.add_alias_field("mass", "Mpeak", units="Msun")
```

On rare occasions, a halo will be missing from the output even though another halo claims it as its descendent. This is usually because the halo has dropped below the minimum mass to be included. In these cases, MORIA will reassign the halo's descendent using the `descendant_index` field (see discussion in here). If `ytree` encounters such a situation, a message like the one below will be printed.

```
>>> t = a[85]
>>> print (t["tree", "Mpeak"])
ytree: [INFO     ] 2021-05-04 15:29:19,723 Reassigning descendent of halo 374749 from␣
↪398837 to 398836.
[1.458e+13 1.422e+13 1.363e+13 1.325e+13 1.295e+13 1.258e+13 1.212e+13 ...
 1.309e+11 1.178e+11 1.178e+11 1.080e+11 9.596e+10 8.397e+10] Msun/h
```

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 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")
```

TreeFrog

TreeFrog generates merger trees primarily for VELOCIraptor halo catalogs. The TreeFrog format consists of a series of HDF5 files. One file contains meta-data for the entire dataset. The other files contain the tree data, split into HDF5 groups corresponding to the original halo catalogs. To load, provide the path to the "foreststats" file, i.e., the one ending in ".hdf5".

```
>>> import ytree
>>> a = ytree.load("treefrog/VELOCIraptor.tree.t4.0-131.walkabletree.sage.forestID.
↪foreststats.hdf5")
```

Merger trees in TreeFrog are organized by *forest*, so printing `a.size` (following the example above) will give the number of forests. Note, however, the id of the root halo for any given forest is not the same as the forest id.

```
>>> my_tree = a[0]
>>> print (my_tree["uid"])
131000000000001
>>> print (my_tree["ForestID"])
104000000011727
```

TreeFrog outputs contain multiple definitions of halo mass, and as such, the field alias "mass" is not defined by default. However, the *alias can be created* if one is preferable. This is also necessary to facilitate *Accessing the Progenitor Lineage of a Tree*.

```
>>> a.add_alias_field("mass", "Mass_200crit", units="Msun")
```

Saved Arbors (ytree format)

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")
```

See *Saving Arbors and Trees* for more information on saving arbors and trees.

2.5.2 Getting Started with Merger Trees

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?

The total number of trees in the arbor can be found using the `size` attribute. As soon as any information about the collection of trees within the loaded dataset is requested, arrays will be created containing the metadata required for generating the root nodes of every tree.

```
>>> 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 the unyt package for symbolic units on NumPy arrays.

```
>>> 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.

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

Array slicing can also be used to select multiple *`TreeNode`* objects. This will return a generator that can be iterated over or cast to a list.

```
>>> every_second_tree = list(a[::2])
>>> print (every_second_tree[0]["mass"])
657410071942446.1 Msun
```

Note, the *`Arbor`* object does not store individual *`TreeNode`* objects, it only generates them. Thus, one must explicitly keep around any *`TreeNode`* object for changes to persist. This is illustrated below:

```
>>> # this will not work
>>> a[0].thing = 5
>>> print (a[0].thing)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'TreeNode' object has no attribute 'thing'
>>> # this will work
>>> my_tree = a[0]
>>> my_tree.thing = 5
>>> print (my_tree.thing)
5
```

The only exception to this is computing the number of nodes in a tree. This information will be propagated back to the `Arbor` as it can be expensive to compute for large trees.

```
>>> my_tree = a[0]
print (my_tree.tree_size) # call function to calculate tree size
691
>>> new_tree = a[0]
print (new_tree.tree_size) # retrieved from a cache
691
```

Accessing the Nodes in a Tree or Forest

A node is defined as a single halo at a single time in a merger tree. Throughout these docs, the words halo and node are used interchangeably. Nodes in a given tree can be accessed in three different ways: by *Accessing All Nodes in a Tree*, *Accessing All Nodes in a Forest*, or *Accessing the Progenitor Lineage of a Tree*. Each of these will return a generator of `TreeNode` objects or field values for all `TreeNode` objects in the tree, forest, or progenitor line. To get a specific node from a tree, see *Accessing a Single Node in a Tree*.

Note: Access by forest is supported even for datasets that do not group trees by forest. If you have no requirement for the order in which nodes are to be returned, then access by forest is recommended as it will be considerably faster than access by tree. Access by tree is effectively a depth-first walk through the tree. This requires additional data structures to be built, whereas forest access does not.

Accessing All Nodes in a Tree

The full lineage of the tree can be accessed by querying any `TreeNode` with the `tree` keyword. As of `ytree` version 3.0, this returns a generator that can be used to loop through all nodes in the tree.

```
>>> print (my_tree["tree"])
<generator object TreeNode._tree_nodes at 0x11bbc1f20>
>>> # loop over nodes
>>> for my_node in my_tree["tree"]:
...     print (my_node, my_node["mass"])
TreeNode[12900] 657410100000000.0 Msun
TreeNode[12539] 657410100000000.0 Msun
TreeNode[12166] 653956900000000.0 Msun
TreeNode[11796] 650071960000000.0 Msun
...
```

To store all the nodes in a single structure, convert it to a list:

```
>>> print (list(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
```

The above examples will work for any halo in the tree, not just the final halo. The full tree leading up to any given halo can be accessed in the same way.

```
>>> tree_nodes = list(my_tree["tree"])
>>> # start with the 3rd halo in the above tree
>>> sub_tree = tree_nodes[2]
>>> print (list(sub_tree["tree"]))
[TreeNode[12166], TreeNode[11796], TreeNode[11431], TreeNode[11077], ...
 TreeNode[591]]
>>> print (sub_tree["tree", "virial_radius"])
[2301.4316  2311.4763  2313.993   2331.413   2345.5454  2349.918 ...
 434.59857  410.13382  411.25757] kpc
```

Accessing All Nodes in a Forest

The *Consistent-Trees-HDF5*, *LHaloTree*, *LHaloTree-HDF5*, *MORIA*, *TreeFrog* formats provide access to halos grouped by forest. A forest is a group of trees with halos that interact in a non-merging way through processes like fly-bys.

```
>>> import ytree
>>> a = ytree.load("consistent_trees_hdf5/soa/forest.h5",
...                access="forest")
>>> my_forest = a[0]
>>> # all halos in the forest
>>> print (list(my_forest["forest"]))
[TreeNode[90049568], TreeNode[88202573], TreeNode[86292249], ...
 TreeNode[9272027], TreeNode[7435733], TreeNode[5768880]]
>>> # all halo masses in forest
>>> print (my_forest["forest", "mass"])
[3.38352524e+11 3.34071450e+11 3.34071450e+11 3.31709477e+11 ...
 7.24092117e+09 4.34455270e+09] Msun
```

To find all the roots in that forest, i.e., the roots of all individual trees contained, see *Accessing the Root Nodes of a Forest*.

Accessing a Halo's Ancestors and Descendent

The direct ancestors of any `TreeNode` object can be accessed through the `ancestors` attribute.

```
>>> my_ancestors = list(my_tree.ancestors)
>>> print (my_ancestors)
[TreeNode[12539]]
```

A halo's descendent can be accessed in a similar fashion.

```
>>> print (my_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. Just as above, this returns a generator of *TreeNode* objects.

```
>>> print (list(my_tree["prog"]))
[TreeNode[12900], TreeNode[12539], TreeNode[12166], TreeNode[11796], ...
 TreeNode[62]]
```

Fields for the main progenitors can be accessed just like for the whole tree.

```
>>> 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
```

Progenitor lists and fields can be accessed for any halo in the tree.

```
>>> tree_nodes = list(my_tree["tree"])
>>> # pick a random halo in the tree
>>> my_halo = tree_nodes[42]
>>> print (list(my_halo["prog"]))
[TreeNode[588], TreeNode[446], TreeNode[317], TreeNode[200], TreeNode[105],
 TreeNode[62]]
>>> print (my_halo["prog", "virial_radius"])
[1404.1354 1381.4087 1392.2404 1363.2145 1310.3842 1258.0159] kpc
```

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")
>>> my_tree = a[0]
>>> print (list(my_tree["prog"]))
```

Accessing a Single Node in a Tree

The *get_node* functions can be used to retrieve a single node from the forest, tree, or progenitor lists.

```
>>> my_tree = a[0]
>>> my_node = my_tree.get_node("forest", 5)
```

This function can be called for any node in a tree. For selection by "tree" or "prog", the index of the node returned will be relative to the calling node, i.e., calling with 0 will return the original node. For selection by "forest", the index is the absolute index within the entire forest and not relative to the calling node.

Accessing the Leaf Nodes of a Tree

The leaf nodes of a tree are the nodes with no ancestors. These are the very first halos to form. Accessing them for any tree can be useful for semi-analytical models or any framework where you want to start at the origins of a halo and work forward in time. The *get_leaf_nodes* function will return a generator of all leaf nodes of a tree's forest, tree, or progenitor lists.

```
>>> my_tree = a[0]
>>> my_leaves = my_tree.get_leaf_nodes(selector="forest")
>>> for my_leaf in my_leaves:
...     print (my_leaf)
```

Similar to the *get_node* function, calling *get_leaf_nodes* with `selector` set to "tree" or "prog" will return only leaf nodes from the tree for which the calling node is the head. With `selector` set to "forest", the resulting leaf nodes will be all the leaf nodes in the forest, regardless of the calling node.

Accessing the Root Nodes of a Forest

A forest can have multiple root nodes, i.e., nodes that have no descendent. The *get_root_nodes* function will return a generator of all the root nodes in the forest. This function can be called from any tree within a forest.

```
>>> my_tree = a[0]
>>> my_roots = my_tree.get_root_nodes()
>>> for my_root in my_roots:
...     print (my_root)
```

2.5.3 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*.

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

2.5.4 Searching Through Merger Trees (Accessing Like a Database)

There are a couple different ways to search through a merger tree dataset to find halos meeting various criteria, similar to the type of selection done with a relational database. The method discussed in *Select Halos* can be used with all data loadable with `ytree`, while the one described in *Select Halos with yt* is only available for *Saved Arbors (ytree format)*.

Select Halos

The *select_halos* function can be used to search the *Arbor* for halos matching a specific set of criteria.

```
>>> halos = list(a.select_halos("tree['forest', 'mass'].to('Msun') > 5e11"))
Selecting halos (found 3): 100%|| 32/32 [00:00<00:00, 107.70it/s]
>>> print (halos)
[TreeNode[1457223360], TreeNode[1457381406], TreeNode[1420495006]]
```

The *select_halos* function will return a generator of *TreeNode* objects that can be iterated over or cast to a list, as above. The function will return halos as they are found so the user does not have to wait until the end to begin working with them. The progress bar will continually update to report the number of matches found.

The selection criteria string should be designed to `eval` correctly with a *TreeNode* object, named "tree". More complex criteria can be supplied using & and |.

```
>>> for halo in a.select_halos("(tree['tree', 'mass'].to('Msun') > 2e11) & (tree['tree
↪', 'redshift'] < 0.2)"):
...     progenitor_pos = halo["prog", "position"]
Selecting halos (found 69): 100%|| 32/32 [00:01<00:00, 22.50it/s]
```

Select Halos with yt

Note: This functionality only works with *Saved Arbors (ytree format)*. You will need to *save your data in the ytree format*.

The *get_yt_selection* function provides enhanced functionality beyond the capabilities of *select_halos* by loading the dataset into yt. Given search criteria, *get_yt_selection* will return a `YTCutRegion` data container that can then be queried to get the value of any field for all halos meeting the criteria. This `YTCutRegion` can then be used to *generate tree nodes* or *query fields*.

Creating the Selection

Search criteria can be provided using a series of keywords: `above`, `below`, `equal`, and `about`.

```
>>> import ytree
>>> a = ytree.load("arbor/arbor.h5")
>>> selection = a.get_yt_selection(,
...     above=[("mass", 1e13, "Msun"),
...            ("redshift", 0.5)])
```

An individual criterion should be expressed as a tuple (e.g., `(field, value, <units>)`), and the above keywords accept a list of those tuples. The criteria keywords can be given together and the halos must meet all criteria, i.e., the criteria are combined with an AND operator.

```
>>> selection = a.get_yt_selection(
...     below=[("mass", 1e13, "Msun")],
...     above=[("redshift", 1)])
```

For more complex search criteria, a cut region conditional string can be provided instead. These should be of the form described in Cut Regions. These cannot not be given with any of the previously mentioned keywords.

```
>>> selection = a.get_yt_selection(
...     conditionals=['obj["halos", "mass"] > 1e12'])
```

Querying the Selection

The selection object returned by *get_yt_selection* can then be queried to get field values for all matching halos. Fields should be queried as `("halos", <field name>)`.

```
>>> # halos with masses of 1e14 Msun +/- 5%
>>> selection = a.get_yt_selection(
        about=[("mass", 1e14, "Msun", 0.05)])

>>> print (selection["halos", "redshift"])
[0.82939091 0.97172537 1.02453741 0.31893065 0.74571856 0.97172537 ...
 0.50455122 0.53499009 0.18907477 0.29567248 0.31893065] dimensionless
```

Getting Halos from the Selection

The *get_nodes_from_selection* function will return a generator of *TreeNode* objects for all halos contained within the selection.

```
>>> # halos with masses of 1e14 Msun +/- 5%
>>> selection = a.get_yt_selection(
        about=[("mass", 1e14, "Msun", 0.05)])

>>> for node in a.get_nodes_from_selection(selector):
...     print (node["prog", "mass"])
```

This function can generate *TreeNode* objects for *any yt data container*.

2.5.5 Halos and Fields from yt Data Containers

Note: This functionality only works with *Saved Arbors (ytree format)*. You will need to *save your data in the ytree format*.

For merger tree data in the *ytree format*, the *ytds* attribute provides access to the data as a yt dataset. This allows one to analyze the entire dataset using the full range of functionality provided by `yt`. In this way, a merger tree dataset is very much like any particle dataset, where each particle represent a halo at a single time. For example, this makes it possible to select halos within geometric data containers, like spheres or regions.

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

(continues on next page)

(continued from previous page)

```
>>> ds = a.ytds
>>> sphere = ds.sphere(ds.domain_center, (5, "Mpc"))
>>> print (sphere["halos", "mass"])
```

These data containers can then be given to the *get_nodes_from_selection* function to *get the tree nodes* for all halos within the container.

```
>>> sphere = ds.sphere(ds.domain_center, (5, "Mpc"))
>>> for node in a.get_nodes_from_selection(sphere):
...     print (node["position"])
```

2.6 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*.

2.6.1 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']}
```

2.6.2 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', ...]
```

2.6.3 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
```

2.6.4 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(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 take two arguments. The first is a dictionary that will contain basic information about the field, such as its name. The second argument represents the data container for which the field will be defined. It can be used to access field data for any other available field. This argument will also have access to the parent *Arbor* as `data.arbor`.

2.6.5 Vector Fields

For fields that have x, y, and z components, such as position, velocity, and angular momentum, a single field can be queried to return an array with all the components. For example, for fields named "position_x", "position_y", and "position_z", the field "position" will return the full vector.

```
>>> print (a["position"])
[[0.0440018, 0.0672202, 0.9569643],
 [0.7383264, 0.1961563, 0.0238852],
 [0.7042797, 0.6165487, 0.500576 ],
 ...
 [0.1822363, 0.1324423, 0.1722414],
 [0.8649974, 0.4718005, 0.7349876]]) unitary
```

A list of defined vector fields can be seen by doing:

```
>>> print (a.field_info.vector_fields)
('position', 'velocity', 'angular_momentum')
```

For all vector fields, a "_magnitude" field also exists, defined as the quadrature sum of the components.

```
>>> print (a["velocity_magnitude"])
[ 488.26936644  121.97143067  146.81450507, ...
  200.74057711  166.13782652  529.7336846 ] km/s
```

Only specifically registered fields will be available as vector fields. For example, saved *Analysis Fields* with x,y,z components will not automatically be available. However, vector fields can be created with the *add_vector_field* function.

```
>>> a.add_vector_field("thing")
```

The above example assumes that fields named "thing_x", "thing_y", and "thing_z" already exist.

2.6.6 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")
>>> my_tree = a[0]
>>> print (my_tree["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 halo in my_tree["tree"]:
...     halo["saucer_sections"] = np.random.random() # complicated analysis
...
>>> print (my_tree["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 saved when the *TreeNode* objects that have been analyzed are saved with *save_arbor* or *save_tree*.

```
>>> my_trees = list(a[:]) # all trees
>>> for my_tree in my_trees:
...     # do analysis...
>>> a.save_arbor(trees=my_trees)
```

Note that we do `my_trees = list(a[:])` and not just `my_trees = a[:]`. This is because `a[:]` is a generator that will return a new set of trees each time. The newly generated trees will not retain changes made to any analysis fields. Thus, we must use `list(a[:])` to explicitly store a list of trees.

Re-saving Analysis Fields

All analysis fields are saved to sidecar files with the "-analysis" keyword appended to them. They can be altered and the arbor re-saved as many times as you like. In the very specific case of re-saving all trees and not providing a new filename or custom list of fields (as in the example above), analysis fields will be saved in place (i.e., over-writing the "-analysis" files). The conventional on-disk fields will not be re-saved as they cannot be altered.

2.7 Plotting Merger Trees

Some relatively simple visualizations of merger trees can be made with the *TreePlot* command.

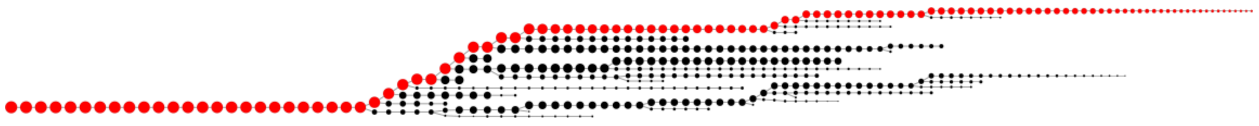
2.7.1 Additional Dependencies

Making merger tree plots with `ytree` requires the pydot and graphviz packages. `pydot` can be installed with `pip` and the graphviz website provides a number of installation options.

2.7.2 Making Tree Plots

The *TreePlot* command can be used to create a digraph depicting halos as filled circles with sizes proportional to their mass. The main progenitor line will be colored red.

```
>>> import ytree
>>> a = ytree.load("ahf_halos/snap_N64L16_000.parameter",
...                hubble_constant=0.7)
>>> p = ytree.TreePlot(a[0], dot_kwargs={'rankdir': 'LR', 'size': '"12,4"'})
>>> p.save('tree.png')
```

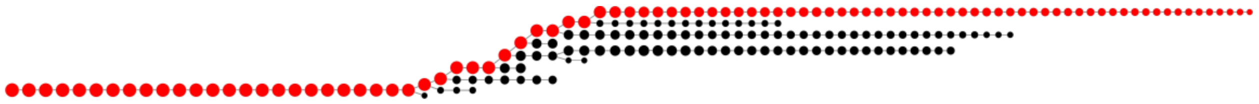


Plot Modifications

Four *TreePlot* attributes can be set to modify the default plotting behavior. These are:

- *size_field*: The field to determine the size of each circle. Default: 'mass'.
- *size_log*: Whether to scale circle sizes based on log of size field. Default: True.
- *min_mass*: The minimum halo mass to be included in the plot. If given as a float, units are assumed to be Msun. Default: None.
- *min_mass_ratio*: The minimum ratio between a halo's mass and the mass of the main halo to be included in the plot. Default: None.

```
>>> import ytree
>>> a = ytree.load("ahf_halos/snap_N64L16_000.parameter",
...                hubble_constant=0.7)
>>> p = ytree.TreePlot(a[0], dot_kwargs={'rankdir': 'LR', 'size': '"12,4"'})
>>> p.min_mass_ratio = 0.01
>>> p.save('tree_small.png')
```



Customizing Node Appearance

The appearance of the nodes can be customized by providing a function that returns a dictionary of keywords that will be used to create the `pydot` node. This should accept a single argument that is a *TreeNode* object representing the halo to be plotted. For example, the following function will add labels of the halo id and mass and make the node shape square. It will also color the most massive progenitor red.

```
def my_node(halo):
    prog = list(halo.find_root()['prog', 'uid'])
    if halo['uid'] in prog:
        color = 'red'
    else:
        color = 'black'

    label = \
    """
    id: %d
    mass: %.2e Msun
    """ % (halo['uid'], halo['mass'].to('Msun'))

```

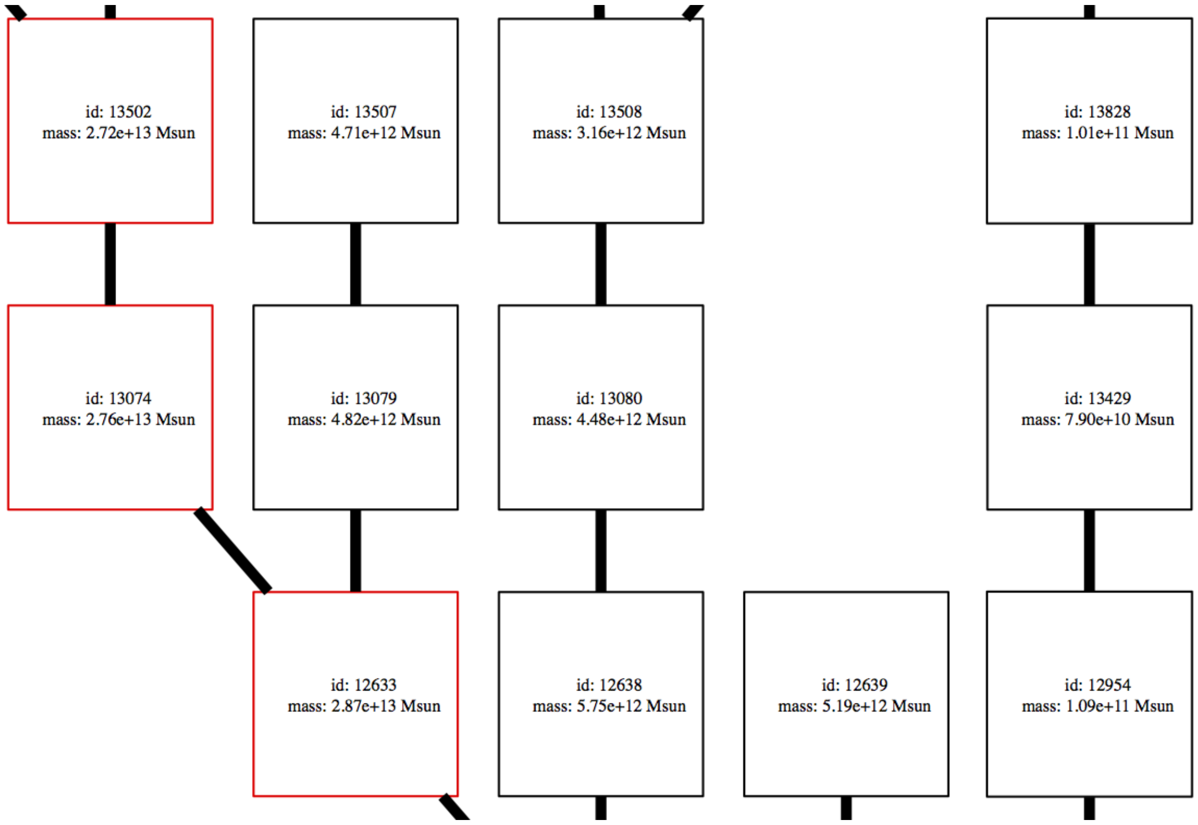
(continues on next page)

(continued from previous page)

```
    my_kwargs = {"label": label, "fontsize": 8,
                 "shape": "square", "color": color}
    return my_kwargs
```

This function is then provided with the *node_function* keyword.

```
>>> p = ytree.TreePlot(tree, dot_kwargs={'rankdir': "BT"},
...                    node_function=my_node)
>>> p.save('tree_custom_node.png')
```



Customizing Edge Appearance

The edges of the plot are the lines connecting each of the nodes. Similar to the nodes, their appearance can be customized by providing a function that returns a dictionary of keywords that will be used to create the `pydot` edge. This should accept two *`TreeNode`* arguments representing the ancestor and descendent halos being connected by the edge. The example below colors the edges blue when the descendent is less massive than its ancestor and green when the descendent is more than 10 times more massive than its ancestor.

```
def my_edge(ancestor, descendent):
    if descendent['mass'] < ancestor['mass']:
        color = 'blue'
    elif descendent['mass'] / ancestor['mass'] > 10:
        color = 'green'
    else:
        color = 'black'

```

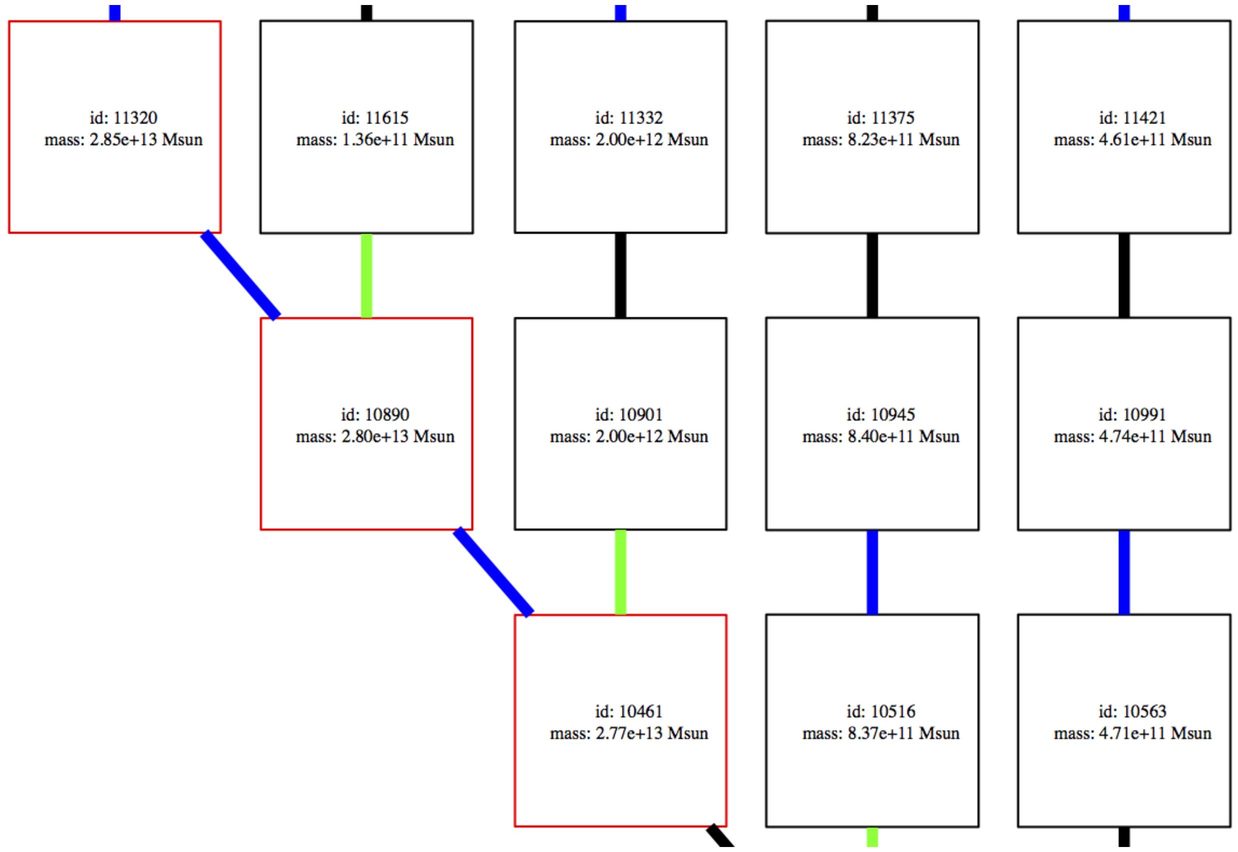
(continues on next page)

(continued from previous page)

```
    my_kwargs = {"color": color, "penwidth": 5}
    return my_kwargs
```

This function is then provided with the *edge_function* keyword.

```
>>> p = ytree.TreePlot(tree, dot_kwargs={'rankdir': "BT"},
...                    node_function=my_node,
...                    edge_function=my_edge)
>>> p.save('tree_custom_edge.png')
```



Supported Output Formats

Plots can be saved to any format supported by `graphviz` by giving a filename with the appropriate extension. See here for a list of currently supported formats.

2.8 Parallel Computing with ytree

`ytree` provides functions for iterating over trees and nodes in parallel. Underneath, they make use of the `parallel_objects` function in `yt`. This functionality is built on MPI, so it can be used to parallelize analysis across multiple nodes of a distributed computing system.

Note: Before reading this section, consult the Parallel Computation With yt section of the `yt` documentation to learn how to configure `yt` for running in parallel.

2.8.1 Enabling Parallelism and Running in Parallel

All parallel computation in `yt` (and hence, `ytree`) begins by importing `yt` and calling the `enable_parallelism` function.

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

In all cases, scripts must be run with `mpirun` to work in parallel. For example, to run on 4 processors, do:

```
$ mpirun -np 4 python my_analysis.py
```

where “my_analysis.py” is the name of the script.

2.8.2 Parallel Iterators

The three parallel iterator functions discussed below are designed to work in conjunction with analysis that makes use of *Analysis Fields*. Minimally, they can be used to iterate over trees and nodes in parallel, but their main advantage is that they will handle the gathering and organization of new analysis field values so they can be properly saved and reloaded. The most efficient function to use will depend on the nature of your analysis.

In the examples below, the “analysis” performed will be facilitated through an *analysis field*, called “test_field”. The following should be assumed to happen before all the examples.

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

a = ytree.load("arbor/arbor.h5")
if "test_field" not in a.field_list:
    a.add_analysis_field("test_field", default=-1, units="Msun")
```

Parallelizing over Trees

The `parallel_trees` function will distribute a list of trees to be analyzed over all available processors. Each processor will work on a single tree in serial.

```
trees = list(a[:])
for tree in ytree.parallel_trees(trees):
    for node in tree["forest"]:
        node["test_field"] = 2 * node["mass"] # this is our analysis
```

At the end of the outer loop, the new values for “test_field” will be collected on the root process (i.e., the process with rank 0) and the arbor will be saved with `save_arbor`. No additional code is required for the new analysis field values to be collected.

By default, each processor will be allocated an equal number of trees. However, this can lead to an unbalanced load if the amount of work varies significantly for each tree. By including the `dynamic=True` keyword, trees will be allocated using a task queue, where each processor is only given another tree after it finishes one. Note, though, that the total number of working processes is one fewer than the number being run with as one will act as the server for the task queue.

```
trees = list(a[:])
for tree in ytree.parallel_trees(trees, dynamic=True):
    for node in tree["forest"]:
        node["test_field"] = 2 * node["mass"] # this is our analysis
```

For various reasons, it may be useful to save results after a certain number of loop iterations rather than only once at the very end. The analysis may take a long time, requiring scripts to be restarted, or keeping results for many trees in memory may be prohibitive. The `save_every` keyword can be used to specify a number of iterations before results are saved. The example below will save results every 8 iterations.

```
trees = list(a[:])
for tree in ytree.parallel_trees(trees, save_every=8):
    for node in tree["forest"]:
        node["test_field"] = 2 * node["mass"] # this is our analysis
```

The default behavior will allocate a tree to a single processor. To allocate more than one processor to each tree, the `njobs` keyword can be used to set the total number of process groups for the loop. For example, if running with 8 total processors, setting `njobs=4` will create 4 groups of 2 processors each.

```
trees = list(a[:])
for tree in ytree.parallel_trees(trees, njobs=4):
    for node in tree["forest"]:
        if yt.is_root():
            node["test_field"] = 2 * node["mass"] # this is our analysis
```

The `is_root` function can be used to determine which process is the root in a group. Only the results recorded by the root process will be collected. In the example above, it is up to the user to properly manage the parallelism within the loop.

Parallelizing over Nodes in a Single Tree

The method presented above in *Parallelizing over Trees* works best when the work done on each node in a tree is small compared to the total number of trees. If the opposite is true, and either the total number of trees is small or the work done on each node is expensive, then it may be better to parallelize over the nodes in a single tree using the *`parallel_tree_nodes`* function. The previous example is parallelized over nodes in a tree in the following way.

```
trees = list(a[:])
for tree in trees:
    for node in ytree.parallel_tree_nodes(tree):
        node["test_field"] = 2 * node["mass"]

if yt.is_root():
    a.save_arbor(trees=trees)
```

Unlike the *`parallel_trees`* and *`parallel_nodes`* functions, no saving occurs automatically. Hence, the results must be saved manually, as in the above example.

The `group` keyword can be set to `forest` (the default), `tree`, or `prog` to control which nodes of the tree are looped over. The `dynamic` and `njobs` keywords have similar behavior as in *Parallelizing over Trees*.

Parallelizing over Nodes in a List of Trees

The previous two examples use a nested loop structure, parallelizing either the outer loop over trees or the inner loop over nodes in a given tree. The `parallel_nodes` function combines these into a single iterator capable of adding parallelism to either the loop over trees, nodes in a tree, or both. With this function, the same example from above becomes:

```
trees = list(a[:])
for node in ytree.parallel_nodes(trees):
    node["test_field"] = 2 * node["mass"]
```

New analysis field values are collected and saved automatically as with the `parallel_trees` function. Similar to `parallel_trees`, the `save_every` keyword can be used to control the number of full trees to be completed before saving results. As well, the `group` keyword can be used to control the nodes iterated over in a tree, similar to how it works in `parallel_tree_nodes`. You will likely be unsurprised to learn that the `parallel_nodes` function is implemented using nested calls to `parallel_trees` and `parallel_tree_nodes`.

The `dynamic` and `njobs` keywords also work similarly, only that they must be specified as tuples of length 2, where the first values control the loop over trees and the second values control the loop over nodes in a tree. Using this, it is possible to enable task queues for both loops (trees and nodes), as in the following example.

```
trees = list(a[:])
for node in ytree.parallel_nodes(trees, save_every=8,
                                 njobs=(3, 0),
                                 dynamic=(True, True)):
    node["test_field"] = 2 * node["mass"]
```

If the above example is run with 13 processors, the result will be a task queue with 3 process groups of 4 processors each. Each of those process groups will work on a single tree using its own task queue, consisting of 1 server process and 3 worker processes. What a world we live in.

2.9 Analyzing Merger Trees

This section describes the preferred method for performing analysis on merger trees that results in modification of the dataset (e.g., by creating or altering *Analysis Fields*) or writing additional files. The end of this section, *Putting it all Together: Parallel Analysis*, illustrates an analysis workflow that will run in parallel.

When performing the same analysis on a large number of items, we often think in terms of creating an “analysis pipeline”, where a series of discrete actions, including deciding whether to skip a given item, are embedded within a loop over all the items to analyze. For merger trees, this may look something like the following:

```
import ytree

a = ytree.load(...)

trees = list(a[:])
for tree in trees:
    for node in tree["forest"]:

        # only analyze above some minimum mass
        if node["mass"] < a.quan(1e11, "Msun"):
            continue

        # get simulation snapshot associated with this halo
        snap_fn = get_filename_from_redshift(node["redshift"])
```

(continues on next page)

(continued from previous page)

```
        ds = yt.load(snap_fn)

        # get sphere using halo's center and radius
        center = node["position"].to("unitary")
        radius = node["virial_radius"].to("unitary")
        sp = ds.sphere((center, "unitary"), (radius, "unitary"))

        # calculate gas mass and save to field
        node["gas_mass"] = sp.quantities.total_quantity(("gas", "mass"))

        # make a projection and save an image
        p = yt.ProjectionPlot(ds, "x", ("gas", "density"),
                              data_source=sp, center=sp.center,
                              width=2*sp.width)
        p.save("my_analysis/projections/")
```

There are a few disadvantages of this approach. The inner loop is very long. It can be difficult to understand the full set of actions, especially if you weren't the one who wrote it. If there is a section you no longer want to do, a whole block of code needs to be commented out or removed, and it may be tricky to tell if doing that will break something else. Putting the operations into functions will make this simpler, but it can still make for a large code block in the inner loop. As well, if the structure of the loops over trees or nodes is more complicated than the above, there is potential for the code to be non-trivial to digest.

2.9.1 The AnalysisPipeline

The *AnalysisPipeline* allows you to design the analysis workflow in advance and then use it to process a tree or node with a single function call. Skipping straight to the end, the loop from above will take the form:

```
for tree in trees:
    for node in tree["forest"]:
        ap.process_target(node)
```

In the above example, "ap" is some *AnalysisPipeline* object that we have defined earlier. We will now take a closer look at how to design a workflow using this method.

Creating an AnalysisPipeline

An *AnalysisPipeline* is instantiated with no arguments. Only an optional output directory inside which new files will be written can be specified with the `output_dir` keyword.

```
import ytree

ap = ytree.AnalysisPipeline(output_dir="my_analysis")
```

The output directory will be created automatically if it does not already exist.

Creating Pipeline Operations

An analysis pipeline is assembled by creating functions that accept a single *TreeNode* as an argument.

```
def say_hello(node):
    print (f"This is node {node}! I will now be analyzed.")
```

This function can now be added to an existing pipeline with the *add_operation* function.

```
ap.add_operation(say_hello)
```

Now, when the *process_target* function is called with a *TreeNode* object, the `say_hello` function will be called with that *TreeNode*. Any additional calls to *add_operation* will result in those functions also being called with that *TreeNode* in the same order.

Adding Extra Function Arguments

Functions can take additional arguments and keyword arguments as well.

```
def print_field_value(node, field, units=None):
    val = node[field]
    if units is not None:
        val.convert_to_units(units)
    print (f"Value of {field} for node {node} is {val}.")
```

The additional arguments and keyword arguments are then provided when calling *add_operation*.

```
ap.add_operation(print_field_value, "mass")
ap.add_operation(print_field_value, "virial_radius", units="kpc/h")
```

Organizing File Output by Operation

In the same way that the *AnalysisPipeline* object accepts an `output_dir` keyword, analysis functions can also accept an `output_dir` keyword.

```
def save_something(node, output_dir=None):
    # make an HDF5 file named by the unique node ID
    filename = f"node_{node.uid}.h5"
    if output_dir is not None:
        filename = os.path.join(output_dir, filename)

    # do some stuff...

# meanwhile, back in the pipeline...
ap.add_operation(save_something, output_dir="images")
```

This `output_dir` keyword will be intercepted by the *AnalysisPipeline* object to ensure that the directory gets created if it does not already exist. Additionally, if an `output_dir` keyword was given when the *AnalysisPipeline* was created, as in the example above, the directory associated with the function will be appended to that. Following the examples here, the resulting directory would be "my_analysis/images", and the code above will correctly save to that location.

Using a Function as a Filter

Making an analysis function return `True` or `False` allows it to act as a filter. If a function returns `False`, then any additional operations defined in the pipeline will not be performed. For example, we might create a mass filter like this:

```
def minimum_mass(node, value):
    return node["mass"] >= value

# later, in the pipeline
ap.add_operation(minimum_mass, a.quan(1e11, "Msun"))
```

Modifying a Node

There may be occasions where you want to pass local variables or objects around from one function to the next. The easiest way to do this is by attaching them to the `TreeNode` object itself as an attribute. For example, say we have a function that returns a simulation snapshot loaded with `yt` as a function of redshift. We might do something like the the following to then pass it to another function which creates a `yt` sphere.

```
def get_yt_dataset(node):
    # assume you have something like this
    filename = get_filename_from_redshift(node["redshift"])
    # attach it to the node for later use
    node.ds = yt.load(filename)

def get_yt_sphere(node):
    # this works if get_yt_dataset has been called first
    ds = node.ds

    center = node["position"].to("unitary")
    radius = node["virial_radius"].to("unitary")
    node.sphere = ds.sphere((center, "unitary"), (radius, "unitary"))
```

Then, we can add these to the pipeline such that a later function can use the sphere.

```
ap.add_operation(get_yt_dataset)
ap.add_operation(get_yt_sphere)
```

To clean things up, we can make a function to remove attributes and add it to the end of the pipeline.

```
def delete_attributes(node, attributes):
    for attr in attributes:
        delattr(node, attr)

# later, in the pipeline
ap.add_operation(delete_attributes, ["ds", "sphere"])
```

Running the Pipeline

Once the pipeline has been defined through calls to `add_operation`, it is now only a matter of looping over the nodes we want to analyze and calling `process_target` with them.

```
for tree in trees:
    for node in tree["forest"]:
        ap.process_target(node)
```

Depending on what you want to do, you may want to call `process_target` with an entire tree and skip the inner loop. After all, a tree in this context is just another `TreeNode` object, only one that has no descendent.

Creating a Analysis Recipe

Through the previous examples, we have designed a workflow by defining functions and adding them to our pipeline in the order we want them to be called. Has it resulted in fewer lines of code? No. But it has allowed us to construct a workflow out of a series of reusable parts, so the creation of future pipelines will certainly involve fewer lines of code. It is also possible to define a more complex series of operations as a "recipe" that can be added in one go to the pipeline using the *add_recipe* function. A recipe should be a function that, minimally, accepts an *AnalysisPipeline* object as the first argument, but can also accept more. Below, we will define a recipe for calculating the gas mass for a halo. For our purposes, assume the functions we created earlier exist here.

```
def calculate_gas_mass(node):
    sphere = node.sphere
    node["gas_mass"] = sphere.quantities.total_quantity(("gas", "mass"))

def gas_mass_recipe(pipeline):
    pipeline.add_operation(get_yt_dataset)
    pipeline.add_operation(get_yt_sphere)
    pipeline.add_operation(calculate_gas_mass)
    pipeline.add_operation(delete_attributes, ["ds", "sphere"])
```

Now, our entire analysis pipeline design can look like this.

```
ap = ytree.AnalysisPipeline()
ap.add_recipe(gas_mass_recipe)
```

See the *add_recipe* docstring for an example of including additional function arguments.

2.9.2 Putting it all Together: Parallel Analysis

To unleash the true power of the *AnalysisPipeline*, run it in parallel using one of the *Parallel Iterators*. See *Parallel Computing with ytree* for more information on using `ytree` on multiple processors.

```
import ytree

a = ytree.load("arbor/arbor.h5")
if "test_field" not in a.field_list:
    a.add_analysis_field("gas_mass", default=-1, units="Msun")

ap = ytree.AnalysisPipeline()
ap.add_recipe(gas_mass_recipe)

trees = list(a[:])
for node in ytree.parallel_nodes(trees):
    ap.process_target(node)
```

If you need some inspiration, have a look at some *Example Applications*.

2.10 Example Applications

Below are some examples of things one might want to do with merger trees that demonstrate various `ytree` functions. If you have made something interesting, please add it!

2.10.1 Halo Age (a50)

One way to define the age of a halo is by calculating the scale factor when it reached 50% of its current mass. This is often referred to as “a50”. In the example below, this is calculated by linearly interpolating from the mass of the main progenitor.

```
import numpy as np

def calc_a50(node):
    # main progenitor masses
    pmass = node["prog", "mass"]

    mh = 0.5 * node["mass"]
    m50 = pmass <= mh

    if not m50.any():
        ah = node["scale_factor"]
    else:
        pscale = node["prog", "scale_factor"]
        # linearly interpolate
        i = np.where(m50)[0][0]
        slope = (pscale[i-1] - pscale[i]) / (pmass[i-1] - pmass[i])
        ah = slope * (mh - pmass[i]) + pscale[i]

    node["a50"] = ah
```

Now we’ll run it using the *The AnalysisPipeline*.

```
>>> import ytree
>>> a = ytree.load("consistent_trees/tree_0_0_0.dat")
>>> a.add_analysis_field("a50", "")

>>> ap = ytree.AnalysisPipeline()
>>> ap.add_operation(calc_a50)

>>> trees = list(a[:])
>>> for tree in trees:
...     ap.process_target(tree)

>>> fn = a.save_arbor(filename="halo_age", trees=trees)
>>> a2 = ytree.load(fn)
>>> print (a2[0]["a50"])
0.57977664
```

2.10.2 Significance

Brought to you by John Wise, a halo’s significance is calculated by recursively summing over all ancestors the mass multiplied by the time between snapshots. When determining the main progenitor of a halo, the significance measure will select for the ancestor with the deeper history instead of just the higher mass. This can be helpful in cases of near 1:1 mergers.

Below, we define a function that calculates the significance for every halo in a single tree.

```
def calc_significance(node):
   if node.descendent is None:
       dt = 0. * node["time"]
```

(continues on next page)

(continued from previous page)

```
    else:
        dt = node.descendent["time"] - node["time"]

    sig = node["mass"] * dt
    if node.ancestors is not None:
        for anc in node.ancestors:
            sig += calc_significance(anc)

    node["significance"] = sig
    return sig
```

Now, we'll use the *The AnalysisPipeline* to calculate the significance for all trees and save a new dataset. After loading the new arbor, we use the `set_selector` function to use the new significance field to determine the progenitor line.

```
>>> a = ytree.load("tiny_ctrees/locations.dat")
>>> a.add_analysis_field("significance", "Msun*Myr")

>>> ap = ytree.AnalysisPipeline()
>>> ap.add_operation(calc_significance)

>>> trees = list(a[:])
>>> for tree in trees:
...     ap.process_target(tree)

>>> fn = a.save_arbor(filename="significance", trees=trees)
>>> a2 = ytree.load(fn)
>>> a2.set_selector("max_field_value", "significance")
>>> prog = list(a2[0]["prog"])
>>> print (prog)
[TreeNode[1457223360], TreeNode[1452164856], TreeNode[1447024182], ...
 TreeNode[6063823], TreeNode[5544219], TreeNode[5057761]]
```

2.11 Community Code of Conduct

ytree is a project by members of the yt community. As such, we stand by the yt Community Code of Conduct.

Below is the ytree version of this code.

# ytree Community Code of Conduct

The community of participants in open source Scientific projects is made up of members from around the globe with a diverse set of skills, personalities, and experiences. It is through these differences that our community experiences success and continued growth. We expect everyone in our community to follow these guidelines when interacting with others both inside and outside of our community. Our goal is to keep ours a positive, inclusive, successful, and growing community.

As members of the community,

- We pledge to treat all people with respect and provide a harassment- and bullying-free environment, regardless of sex, sexual orientation and/or gender identity, disability, physical appearance, body size, race, nationality, ethnicity, and religion. In particular, sexual language and imagery, sexist, racist, or otherwise exclusionary jokes are not appropriate.
- We pledge to respect the work of others by recognizing acknowledgment/citation requests of original authors. As authors, we pledge to be explicit about how we want our own work to be cited or acknowledged.

- We pledge to welcome those interested in joining the community, and realize that including people with a variety of opinions and backgrounds will only serve to enrich our community. In particular, discussions relating to pros/cons of various technologies, programming languages, and so on are welcome, but these should be done with respect, taking proactive measure to ensure that all participants are heard and feel confident that they can freely express their opinions.
- We pledge to welcome questions and answer them respectfully, paying particular attention to those new to the community. We pledge to provide respectful criticisms and feedback in forums, especially in discussion threads resulting from code contributions.
- We pledge to be conscientious of the perceptions of the wider community and to respond to criticism respectfully. We will strive to model behaviors that encourage productive debate and disagreement, both within our community and where we are criticized. We will treat those outside our community with the same respect as people within our community.

We pledge to help the entire community follow the code of conduct, and to not remain silent when we see violations of the code of conduct. We will take action when members of our community violate this code such as contacting the project manager, Britton Smith (brittonsmith@gmail.com). All emails will be treated with the strictest confidence or talking privately with the person.

This code of conduct applies to all community situations online and offline, including mailing lists, forums, social media, conferences, meetings, associated social events, and one-to-one interactions.

This Community Code of Conduct comes the <a href="http://yt-project.org/community.html"> yt Community Code of Conduct</a>, which was adapted from the <a href="http://www.astropy.org/about.html#codeofconduct"> Astropy Community Code of Conduct</a>, which was partially inspired by the PSF code of conduct.

2.12 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 to know more, contact Britton Smith (brittonsmith@gmail.com) or come by the #ytree channel on the yt project Slack.

You can also find help on the yt developers list.

2.13 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`.

2.13.1 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.

2.13.2 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.

Installing Development Dependencies

A number of additional packages are required for testing. These can be installed with pip from within the `ytree` source by doing:

```
$ pip install -e .[dev]
```

To see how these dependencies are defined, have a look at the `extras_require` keyword argument in the `setup.py` file.

Run the Tests

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 ........
```

(continues on next page)

(continued from previous page)

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

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

2.13.3 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/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/frontends/api.py`.

Building Your Frontend

A very good way to build a new frontend is to start with an existing frontend for a similar type of dataset. To see the variety of examples, consult the *Internal Classes* section of the *API Reference*.

To build a new frontend, you will need to make frontend-specific subclasses for a few components. A straightforward way to do this is to start with the script below, loading your data with it. Each line will run correctly after a distinct phase of the implementation is completed. As you progress, the next function needing implemented will raise a `NotImplementedError` exception, indicating what should be done next.

```
import ytree

# Arbor subclass with working _is_valid function
a = ytree.load(<your data>)

# Recognizing the available fields
print (a.field_list)

# Calculate the number of trees in the dataset
print (a.size)

# Create root TreeNode objects
my_tree = a[0]
print (my_tree)

# Query fields for individual trees
print (my_tree['mass'])

# Query fields for a whole tree
print (my_tree['tree', 'mass'])
```

(continues on next page)

(continued from previous page)

```python
# Create TreeNodes for whole tree
for node in my_tree['tree']:
    print (node)

# Query fields for all root nodes
print (a['mass'])

# Putting it all together
a.save_arbor()
```

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:

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

The `_is_valid` Function

Within every *Arbor* subclass should appear a method called `_is_valid`. This function is used by `load` to determine if the provided 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.

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. These include the `consistent-trees`, `consistent-trees-hdf5`, `lhalotree`, and `ytree` frontends.

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. There are also hybrid formats that use both merger tree and halo catalog files together. An example of this is the `ahf` (Amiga Halo Finder) frontend.

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 creating arrays of the data required to build all the root *TreeNode* objects in the *Arbor*. The names of these attributes are declared in the `_node_io_attrs` attribute. For example, the *ConsistentTreesHDF5Arbor* class names three required attributes: `_fi`, the data file number in which this tree lives; `_si`, the starting index of the section in the data array corresponding to this tree; and `_ei`, the ending index in the data 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 of NumPy arrays for each field.

Halo Catalog-style Data

If this is your case, then the rockstar and treefarm 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.

Field Units and Aliases (`fields.py`)

The *FieldInfoContainer* class holds information about field names and units. Your subclass can define two tuples, `known_fields` and `alias_fields`. The `known_fields` tuple is used to set units for fields on disk. This is useful especially if there is no way to get this information from the file. The convention for each entry is (name on disk, units).

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. This is done with the `alias_fields` tuple. The convention for each entry is (alias name, name on disk, field units).

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

class NewCodeFieldInfo(FieldInfoContainer):
    known_fields = (
        # name on disk, units
        ("Mass", "Msun/h"),
        ("PX", "kpc/h"),
    )

    alias_fields = (
        # alias name, name on disk, units for alias
```

(continues on next page)

(continued from previous page)

```
        ("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.

2.14 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. There is a dedicated #ytree channel on the yt project Slack and questions can also be posted to the yt users mailing list. Bugs and feature requests can also be posted on the ytree issues page.

See you out there!

2.15 Citing ytree

If you use ytree in your work, please cite the following:

Smith et al., (2019). ytree: A Python package for analyzing merger trees. Journal of Open Source Software, 4(44), 1881, https://doi.org/10.21105/joss.01881

For BibTeX users:

```
@article{ytree,
  doi = {10.21105/joss.01881},
  url = {https://doi.org/10.21105/joss.01881},
  year  = {2019},
  month = {dec},
  publisher = {The Open Journal},
  volume = {4},
  number = {44},
  pages = {1881},
  author = {Britton D. Smith and Meagan Lang},
  title = {ytree: A Python package for analyzing merger trees},
```

(continues on next page)

(continued from previous page)

```
  journal = {Journal of Open Source Software}
}
```

If you would like to also cite the specific version of ytree used in your work, include the following reference:

```
@software{britton_smith_2021_5336665,
  author       = {Britton Smith and
                  Meagan Lang and
                  Juanjo Bazán},
  title        = {ytree-project/ytree: ytree 3.1 Release},
  month        = aug,
  year         = 2021,
  publisher    = {Zenodo},
  version      = {ytree-3.1.0},
  doi          = {10.5281/zenodo.5336665},
  url          = {https://doi.org/10.5281/zenodo.5336665}
}
```

2.16 Reference

Below are some reference materials for ytree, including API documentation for all available functionality and a log of changes from each stable release.

2.16.1 API Reference

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.
add_alias_field(alias, field[, units, force_add])	Add a field as an alias to another field.
add_analysis_field(name, units[, dtype, default])	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.
add_vector_field(name)	Add vector fields for a set of x,y,z component fields.
save_arbor(**kwargs)	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.
TreeNode(uid[, arbor, root])	Class for objects stored in Arbors.
get_leaf_nodes([selector])	Get all leaf nodes from the tree of which this is the head.
get_root_nodes()	Get all root nodes from the forest to which this node belongs.
get_node(selector, index)	Get a single TreeNode from a tree.
save_tree([filename, fields])	Save the tree to a file.

Continued on next page

Table 1 – continued from previous page

`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.
`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.
`get_yt_selection`([above, below, equal, ...])	Get a selection of halos meeting given criteria.
`get_nodes_from_selection`(container)	Generate TreeNodes from a yt data container.
`ytds`	Load as a yt dataset.

ytree.data_structures.load.load

ytree.data_structures.load.**load**(*filename*, *method=None*, ***kwargs*)

Load an Arbor, determine the type automatically.

Parameters

- **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.
- **kwargs** (*optional, dict*) – Additional keyword arguments are passed to _is_valid and the determined type.

Returns

Return type *Arbor*

Examples

```
>>> import ytree
>>> # saved Arbor (ytree format)
>>> a = ytree.load("arbor/arbor.h5")
>>> # Amiga Halo Finder
>>> a = ytree.load("ahf_halos/snap_N64L16_000.parameter",
...                hubble_constant=0.7)
>>> # consistent-trees
>>> a = ytree.load("tiny_ctrees/locations.dat")
>>> a = ytree.load("consistent_trees/tree_0_0_0.dat")
>>> a = ytree.load("ctrees_hlists/hlists/hlist_0.12521.list")
>>> # consistent-trees-hdf5
>>> a = ytree.load("consistent_trees_hdf5/soa/forest.h5")
>>> # LHaloTree
>>> a = ytree.load("my_halos/trees_063.0")
>>> # LHaloTree-hdf5
>>> a = ytree.load("TNG50-4-Dark/trees_sf1_099.0.hdf5",
```

(continues on next page)

(continued from previous page)

```
...                   box_size=35, hubble_constant=0.6774,
...                   omega_matter=0.3089, omega_lambda=0.6911)
>>> # Moria
>>> a = ytree.load("moria/moria_tree_testsim050.hdf5")
>>> # Rockstar
>>> a = ytree.load("rockstar_halos/out_0.list")
>>> # treefarm
>>> a = ytree.load("my_halos/fof_subhalo_tab_025.0.h5")
>>> # TreeFrog
>>> a = ytree.load("treefrog/VELOCIraptor.tree.t4.0-131.walkabletree.sage.
↪forestID.foreststats.hdf5")
```

ytree.data_structures.arbor.Arbor

class ytree.data_structures.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 unyt_arrays and unyt_quantities 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[, dtype, default])	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.
add_vector_field(name)	Add vector fields for a set of x,y,z component fields.
get_nodes_from_selection(*args, **kwargs)	
get_yt_selection(*args, **kwargs)	
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.
reset_node(tree_node)	Reset all data structures for a single node.
save_arbor(**kwargs)	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.

Attributes

`arr`	Create a unyt_array using the Arbor's unit registry.
`box_size`	The simulation box size.
`field_info`	A dictionary containing information for each available field.
`hubble_constant`	Value of the Hubble parameter.
`is_planted`	Determine if trees have been planted.
`omega_lambda`	
`omega_matter`	
`omega_radiation`	
`quan`	Create a unyt_quantity using the Arbor's unit registry.
`size`	Return total number of trees.
`unit_registry`	Unit system registry.
`ytds`	

ytree.data_structures.arbor.Arbor.add_alias_field

`Arbor.`**`add_alias_field`**(*alias*, *field*, *units=None*, *force_add=True*)

Add a field as an alias to another field.

Parameters

- **alias** (*string*) – Alias name.
- **field** (*string*) – The field to be aliased.
- **units** (*optional, string*) – Units in which the field will be returned.
- **force_add** (*optional, bool*) – If True, add field even if it already exists and warn the user and raise an exception if dependencies do not exist. If False, silently do nothing in both instances. Default: True.

Examples

```
>>> import ytree
>>> a = ytree.load("tree_0_0_0.dat")
>>> # "Mvir" exists on disk
>>> a.add_alias_field("mass", "Mvir", units="Msun")
>>> print (a["mass"])
```

ytree.data_structures.arbor.Arbor.add_analysis_field

`Arbor.`**`add_analysis_field`**(*name*, *units*, *dtype=None*, *default=0*)

Add an empty field to be filled by analysis operations.

Parameters

- **name** (*string*) – Field name.
- **units** (*string*) – Field units.
- **dtype** (*optional, type*) – Data type for field values. If None, the default data type of the arbor is used. Default: None.

- **default** (*optional, numeric*) – Default field value when field is initialized. Default: 0.

Examples

```
>>> import ytree
>>> a = ytree.load("tree_0_0_0.dat")
>>> a.add_analysis_field("robots", "Msun * kpc")
>>> # Set field for some halo.
>>> my_tree = a[0]
>>> my_tree["tree"][7]["robots"] = 1979.816
```

ytree.data_structures.arbor.Arbor.add_derived_field

Arbor.**add_derived_field**(*name, function, units=None, dtype=None, description=None, vector_field=False, force_add=True*)

Add a field that is a function of other fields.

Parameters

- **name** (*string*) – Field name.
- **function** (*callable*) – The function to be called to generate the field. This function should take two arguments, the arbor and the data structure containing the dependent fields. See below for an example.
- **units** (*optional, string*) – The units in which the field will be returned.
- **dtype** (*optional, type*) – The data type of the field array. If none, use the default type set by Arbor._default_dtype.
- **description** (*optional, string*) – A short description of the field.
- **vector_field** (*optional, bool*) – If True, field is an xyz vector. Default: False.
- **force_add** (*optional, bool*) – If True, add field even if it already exists and warn the user and raise an exception if dependencies do not exist. If False, silently do nothing in both instances. Default: True.

Examples

```
>>> import ytree
>>> a = ytree.load("tree_0_0_0.dat")
>>> def _redshift(field, data):
...     return 1. / data["scale"] - 1
...
>>> a.add_derived_field("redshift", _redshift)
>>> print (a["redshift"])
```

ytree.data_structures.arbor.Arbor.add_vector_field

Arbor.**add_vector_field**(*name*)

Add vector fields for a set of x,y,z component fields.

This will add a general vector field that returns the combined x, y, z components as a single Nx3 array. A <field>_magnitude field with the quadrature sum of the components is also added.

Parameters **`name`** (*`string`*) – The name of the field. Component x,y,z fields must exist.

Examples

```
>>> import ytree
>>> a = ytree.load("tree_0_0_0.dat")
>>> for ax in 'xyz':
>>>     a.add_analysis_field(f"thing_{ax}")
>>> fn = a.save_arbor()
>>> a_new = ytree.load(fn)
>>> a_new.add_vector_field("thing")
>>> print (a_new["thing"])
>>> print (a_new["thing_magnitude"])
```

ytree.data_structures.arbor.Arbor.save_arbor

`Arbor.`**`save_arbor`**`(`***kwargs*`)`

Save the arbor to a file.

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

Parameters

- **`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.
- **`trees`** (*`optional, list or array of TreeNodes`*) – If given, only save trees stemming from these nodes. If not provide, all trees will be saved.
- **`max_file_size`** (*`optional, float`*) – The maximum number of nodes saved to a single file. Smaller numbers will result in more files. Performance may change somewhat with different values. Default: 524288 (2^19).

Returns **header_filename** – The filename of the saved arbor.

Return type string

Examples

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

ytree.data_structures.arbor.Arbor.select_halos

`Arbor.`**`select_halos`**(*criteria*, *trees=None*, *select_from=None*, *fields=None*)

Select halos from the arbor based on a set of criteria given as a string.

Halos matching the criteria will be returned through a generator. Matches are returned as soon as they are found, allowing you to begin working with them before the search has completed. The progress bar will update to report the number of matches found as the search progresses.

Parameters

- **`criteria`** (*`string`*) – A string that will eval to a Numpy-like selection operation performed on a TreeNode object called "tree". Example: 'tree["tree", "redshift"] > 1'
- **`trees`** (*`optional, list or array of TreeNodes`*) – A list or array of TreeNode objects in which to search. If none given, the search is performed over the full arbor.
- **`select_from`** (*`deprecated, do not use`*) – This keyword is no longer required and using it does nothing.
- **`fields`** (*`deprecated, do not use`*) – This keyword is no longer required and using it does nothing.

Returns **halos** – A generator yielding all TreeNodes meeting the criteria.

Return type *`TreeNode`* generator

Examples

```
>>> import ytree
>>> a = ytree.load("tree_0_0_0.dat")
>>> for halo in a.select_halos('tree["tree", "redshift"] > 1'):
...     print (halo["mass"])
>>>
>>> halos = list(a.select_halos('tree["prog", "mass"].to("Msun") >= 1e10'))
>>> print (len(halos))
```

ytree.data_structures.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.

Parameters

- **`selector`** (*`string`*) – Name of the selector to be used.
- **`additional arguments and keywords to be provided to`** (*`Any`*) –
- **`selector function should follow.`** (*`the`*) –

Examples

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

ytree.data_structures.tree_node.TreeNode

class ytree.data_structures.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.
clear_fields()	If a root node, delete field data.
find_root()	Find the root node.
get_leaf_nodes([selector])	Get all leaf nodes from the tree of which this is the head.
get_node(selector, index)	Get a single TreeNode from a tree.
get_root_nodes()	Get all root nodes from the forest to which this node belongs.
query(key)	Return field values for this TreeNode, progenitor list, or tree.
save_tree([filename, fields])	Save the tree to a file.
walk_to_root()	Walk descendents until root.

Attributes

ancestors	Return a generator of ancestor nodes.
desc_uids	Array of descendent uids for all nodes in the tree.
descendent	Return the descendent node.
is_root	Is this node the last in the tree?
tree_id	Return the index of this node in a list of all nodes in the tree.
tree_size	Number of nodes in the tree.
uids	Array of uids for all nodes in the tree.

ytree.data_structures.tree_node.TreeNode.get_leaf_nodes

TreeNode.**get_leaf_nodes**(*selector=None*)

Get all leaf nodes from the tree of which this is the head.

This returns a generator of all leaf nodes belonging to this tree. A leaf node is a node that has no ancestors.

Parameters **selector** (*optional, str ("forest", "tree", or "prog")*) – The tree selector from which leaf nodes will be found. If none given, this will be set to "forest" if the calling node is a root node and "tree" otherwise.

Returns **leaf_nodes** – *TreeNode* objects.

Return type a generator of

Examples

```
>>> import ytree
>>> a = ytree.load("tiny_ctrees/locations.dat")
>>> my_tree = a[0]
>>> for leaf in my_tree.get_leaf_nodes():
...     print (leaf["mass"])
```

ytree.data_structures.tree_node.TreeNode.get_root_nodes

TreeNode.**get_root_nodes**()

Get all root nodes from the forest to which this node belongs.

This returns a generator of all root nodes in the forest. A root node is a node that has no descendents.

Returns **root_nodes** – *TreeNode* objects.

Return type a generator of

Examples

```
>>> import ytree
>>> a = ytree.load("consistent_trees_hdf5/soa/forest.h5",
...                access="forest")
>>> my_tree = a[0]
>>> for root in my_tree.get_root_nodes():
...     print (root["mass"])
```

ytree.data_structures.tree_node.TreeNode.get_node

TreeNode.**get_node**(*selector, index*)

Get a single TreeNode from a tree.

Use this to get the nth TreeNode from a forest, tree, or progenitor list for which the calling TreeNode is the head.

Parameters

- **selector** (*str ("forest", "tree", or "prog")*) – The tree selector from which to get the TreeNode. This should be "forest", "tree", or "prog".
- **index** (*int*) – The index of the desired TreeNode in the forest, tree, or progenitor list.

Returns **node**

Return type *TreeNode*

Examples

```
>>> import ytree
>>> a = ytree.load("tiny_ctrees/locations.dat")
>>> my_tree = a[0]
>>> # get 6th TreeNode in the progenitor list
>>> my_node = my_tree.get_node('prog', 5)
```

ytree.data_structures.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.

Parameters

- **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.

Returns **filename** – The filename of the saved arbor.

Return type string

Examples

```
>>> 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.data_structures.tree_node_selector.TreeNodeSelector

class `ytree.data_structures.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.

Parameters

- **ancestors** (*list of TreeNode objects*) – List of TreeNode objects from which to select.
- **function should return a single TreeNode.** (*The*) –

Examples

```
>>> 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*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(function[, args, kwargs])	Initialize self.

ytree.data_structures.tree_node_selector.add_tree_node_selector

ytree.data_structures.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*.

Parameters

- **name** (*string*) – Name of the selector.
- **function** (*callable*) – The associated function.

Examples

```
>>> 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.data_structures.tree_node_selector.max_field_value

ytree.data_structures.tree_node_selector.**max_field_value**(*ancestors*, *field*)
Return the TreeNode with the maximum value of the given field.

Parameters

- **ancestors** (*list of TreeNode objects*) – List of TreeNode objects from which to select.
- **field** (*string*) – Field to be used for selection.

Returns

Return type TreeNode object

ytree.data_structures.tree_node_selector.min_field_value

`ytree.data_structures.tree_node_selector.`**`min_field_value`**(*ancestors*, *field*)

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

Parameters

- **`ancestors`** (*`list of TreeNode objects`*) – List of TreeNode objects from which to select.
- **`field`** (*`string`*) – Field to be used for selection.

Returns

Return type TreeNode object

ytree.frontends.ytree.arbor.YTreeArbor.get_yt_selection

`YTreeArbor.`**`get_yt_selection`**(*above=None*, *below=None*, *equal=None*, *about=None*, *conditionals=None*, *data_source=None*)

Get a selection of halos meeting given criteria.

This function can be used to create database-like queries to search for halos meeting various criteria. It will return a `YTCutRegion` that can be queried to get field values for all halos meeting the selection criteria. The `YTCutRegion` can then be passed to *`get_nodes_from_selection`* to get all the *`TreeNode`* objects that meet the criteria.

If multiple criteria are provided, selected halos must meet all criteria.

To specify a custom data container, use the `ytds` attribute associated with the arbor to access the merger tree data as a yt dataset. For example:

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

Parameters

- **`above`** (*`optional, list of tuples with (field, value, <units>)`*) – Halos meeting a given criterion must have field values at or above the provided limiting value. Each entry in the list must contain the field name, limiting value, and (optionally) units.
- **`below`** (*`optional, list of tuples with (field, value, <units>)`*) – Halos meeting a given criterion must have field values at or below the provided limiting value. Each entry in the list must contain the field name, limiting value, and (optionally) units.
- **`equal`** (*`optional, list of tuples with (field, value, <units>)`*) – Halos meeting a given criterion must have field values equal to the provided value. Each entry in the list must contain the field name, value, and (optionally) units.
- **`about`** (*`optional, list of tuples with (field, value, tolerance, <units>)`*) – Halos meeting a given criterion must have field values within the tolerance of the provided value. Each entry in the list must contain the field name, value, tolerance, and (optionally) units.

- **conditionals** (*optional, list of strings*) – A list of conditionals for constructing a custom `YTCutRegion`. This can be used instead of above/below/equal/about to create more complex selection criteria. See the Cut Regions section in the yt documentation for more information. The conditionals keyword can only be used if none of the first for selection keywords are given.
- **data_source** (optional, `YTDataContainer`) – The source yt data container to be used to make the cut region. If none given, the `all_data` container (i.e., the full dataset) is used.

Returns **cr** – The cut region associated with the provided selection criteria.

Return type `YTCutRegion`

Examples

```
>>> import ytree
>>> a = ytree.load("arbor/arbor.h5")
>>> # select halos above 1e12 Msun at redshift > 0.5
>>> sel = a.get_yt_selection(
...     above=[("mass", 1e13, "Msun"),
...            ("redshift", 0.5)])
>>> print (sel["halos", "mass"])
>>> print (sel["halos", "virial_radius"])
```

```
>>> import ytree
>>> a = ytree.load("arbor/arbor.h5")
>>> # select halos below 1e13 Msun at redshift > 1
>>> sel = a.get_yt_selection(
...     below=[("mass", 1e13, "Msun")],
...     above=[("redshift", 1)])
>>> print (sel["halos", "mass"])
>>> print (sel["halos", "virial_radius"])
```

```
>>> import ytree
>>> a = ytree.load("arbor/arbor.h5")
>>> # select phantom halos (a consistent-trees field)
>>> sel = a.get_yt_selection(equal=[("phantom", 1)])
```

```
>>> import ytree
>>> a = ytree.load("arbor/arbor.h5")
>>> # select halos with vmax of 200 +-10 km/s (i.e., 5%)
>>> sel = a.get_yt_selection(about=[("vmax", 200, "km/s", 0.05)])
```

```
>>> import ytree
>>> a = ytree.load("arbor/arbor.h5")
>>> # use a yt conditional
>>> sel = a.get_yt_selection(
...     conditionals=['obj["halos", "mass"] > 1e12'])
```

```
>>> import ytree
>>> a = ytree.load("arbor/arbor.h5")
>>> # select halos only within a sphere
>>> ds = a.ytds
>>> sphere = ds.sphere(ds.domain_center, (10, Mpc))
>>> sel = a.get_yt_selection(
```

(continues on next page)

(continued from previous page)

```
...     above=[("mass", 1e13)],
...     data_source=sphere)
>>> # get the TreeNodes for the selection
>>> for node in a.get_nodes_from_selection(sel):
...     print (node["mass"])
```

See also:

`select_halos`, *`get_nodes_from_selection`*

ytree.frontends.ytree.arbor.YTreeArbor.get_nodes_from_selection

`YTreeArbor.`**`get_nodes_from_selection`**`(`*container*`)`

Generate TreeNodes from a yt data container.

All halos contained within the data container will be returned as TreeNode objects. This returns a generator that can be iterated over or cast as a list.

Parameters **`container`** (`YTDataContainer`) – Data container, such as a sphere or region, from which nodes will be generated.

Returns **nodes** – The *`TreeNode`* objects contained within the container.

Return type generator

Examples

```
>>> import ytree
>>> a = ytree.load("arbor/arbor.h5")
>>> c = a.arr([0.5, 0.5, 0.5], "unitary")
>>> sphere = a.ytds.sphere(c, (0.1, "unitary"))
>>> for node in a.get_nodes_from_selection(sphere):
...     print (node["mass"])
```

```
>>> import ytree
>>> a = ytree.load("arbor/arbor.h5")
>>> # select halos above 1e12 Msun at redshift > 0.5
>>> sel = a.get_yt_selection(
...     above=[("mass", 1e13, "Msun"),
...            ("redshift", 0.5)])
>>> my_nodes = list(a.get_nodes_from_selection(sel))
```

ytree.frontends.ytree.arbor.YTreeArbor.ytds

`YTreeArbor.`**`ytds`**

Load as a yt dataset.

Merger tree data is loaded as a yt dataset, providing full access to yt functionality. Fields are accessed with the naming convention, (“halos”, <field name>).

Examples

```
>>> import ytree
>>> a = ytree.load("arbor/arbor.h5")
>>>
>>> ds = a.ytds
>>> sphere = ds.sphere(ds.domain_center, (5, "Mpc"))
>>> print (sphere["halos", "mass"])
>>>
>>> for node in a.get_nodes_from_selection(sphere):
...     print (node["position"])
```

Visualizing Merger Trees

Functionality for plotting merger trees.

TreePlot(tree[, dot_kwargs, node_function, . . .])	Make a simple merger tree plot using pydot and graphviz.
save([filename])	Save the merger tree plot.

ytree.visualization.tree_plot.TreePlot

class `ytree.visualization.tree_plot.`**`TreePlot`**(*tree*, *dot_kwargs=None*, *node_function=None*, *edge_function=None*)

Make a simple merger tree plot using pydot and graphviz.

Parameters

- **tree** (merger tree node *TreeNode*) – The merger tree to be plotted.
- **dot_kwargs** (*optional, dict*) – A dictionary of keyword arguments to be passed to pydot.Dot. Default: None.
- **node_function** (*optional, function*) – A function accepting a single argument of a *TreeNode* and returning a dictionary of keywords to be given to pydot for creating the node object on the plot. This can be used to customize the appearance of the nodes. See examples below. Default: None.
- **edge_function** (*optional, function*) – A function accepting two *TreeNode* objects and returning a dictionary of keywords to be given to pydot for creating the edge object on the plot (the lines connecting halos). This can be used to customize the appearance of the edges. See examples below. Default: None.

size_field

The field to determine the size of each circle. Default: 'mass'.

Type str

size_log

Whether to scale circle sizes based on log of size field. Default: True.

Type bool

min_mass

The minimum halo mass to be included in the plot. If given as a float, units are assumed to be Msun. Default: None.

Type float or unyt_quantity

min_mass_ratio
The minimum ratio between a halo's mass and the mass of the main halo to be included in the plot. Default: None.

Type float

Examples

```
>>> import ytree
>>> a = ytree.load("tree_0_0_0.dat")
>>> p = ytree.TreePlot(a[0])
>>> p.min_mass = 1e6 # Msun
>>> p.save()
```

```
>>> # customizing nodes
>>> import ytree
>>> def my_node(halo):
...     label = f"{halo['uid']}"
...     my_kwargs = {"label": label, "fontsize": 8, "shape": "square"}
...     return my_kwargs
>>> a = ytree.load("tree_0_0_0.dat")
>>> p = ytree.TreePlot(a[0], node_function=my_node)
>>> p.save()
```

```
>>> # customizing edges
>>> import ytree
>>> def my_edge(ancestor, descendent):
...     if descendent['mass'] < ancestor['mass']:
...         color = 'blue'
...     else:
...         color = 'black'
...     my_kwargs = {"color": color, "penwidth": 5}
...     return my_kwargs
>>> a = ytree.load("tree_0_0_0.dat")
>>> p = ytree.TreePlot(a[0], edge_function=my_edge)
>>> p.save()
```

__init__(*tree*, *dot_kwargs=None*, *node_function=None*, *edge_function=None*)
Initialize a TreePlot.

Methods

__init__(tree[, dot_kwargs, node_function, . . .])	Initialize a TreePlot.
save([filename])	Save the merger tree plot.

Attributes

min_mass	The minimum halo mass to be included in the plot.
min_mass_ratio	The minimum halo mass to main halo mass.
size_field	The field to determine the size of each circle.

Continued on next page

Table 9 – continued from previous page

size_log	Whether to scale circle sizes based on log of size field.

ytree.visualization.tree_plot.TreePlot.save

TreePlot.**save**(*filename=None*)

Save the merger tree plot.

Parameters **filename** (*optional, str*) – The output filename. If none given, the uid of the head node is used. Default: None.

Examples

```
>>> import ytree
>>> a = ytree.load("tree_0_0_0.dat")
>>> p = ytree.TreePlot(a[0])
>>> p.save('tree.png')
```

Analysis Pipeline

Machinery for creating a pipeline of analysis to be run on halos in a merger tree.

AnalysisPipeline([output_dir])	Initialize an AnalysisPipeline.
add_operation(function, *args, **kwargs)	Add an operation to the AnalysisPipeline.
add_recipe(function, *args, **kwargs)	Add a recipe to the AnalysisPipeline.
process_target(target)	Process a node through the AnalysisPipeline.
AnalysisOperation(function, *args, **kwargs)	An analysis task performed by an AnalysisPipeline.

ytree.analysis.analysis_pipeline.AnalysisPipeline

class ytree.analysis.analysis_pipeline.**AnalysisPipeline**(*output_dir=None*)

Initialize an AnalysisPipeline.

An AnalysisPipeline allows one to create a workflow of analysis to be performed on a node/halo in a tree. This is done by creating functions that minimally accept a node as the first argument and providing these to the AnalysisPipeline in the order they are meant to be run. This makes it straightforward to organize an analysis workflow into a series of distinct, reusable functions.

Parameters **output_dir** (*optional, str*) – Path to a directory into which any files will be saved. The directory will be created if it does not already exist.

Examples

```
>>> import ytree
>>>
>>> def my_analysis(node):
...     node["test_field"] = 2 * node["mass"]
>>>
>>> def minimum_mass(node, value):
```

(continues on next page)

(continued from previous page)

```
...     return node["mass"] > value
>>>
>>> def my_recipe(pipeline):
...     pipeline.add_operation(my_analysis)
>>>
>>> a = ytree.load("arbor/arbor.h5")
>>>
>>> ap = AnalysisPipeline()
>>> # don't analyze halos below 3e11 Msun
>>> ap.add_operation(minimum_mass, 3e11)
>>> ap.add_recipe(my_recipe)
>>>
>>> trees = list(a[:])
>>> for tree in trees:
...     for node in tree["forest"]:
...         ap.process_target(node)
>>>
>>> a.save_arbor(trees=trees)
```

`__init__`(*output_dir=None*)
Initialize self. See help(type(self)) for accurate signature.

Methods

`__init__`([output_dir])	Initialize self.
`add_operation`(function, *args, **kwargs)	Add an operation to the AnalysisPipeline.
`add_recipe`(function, *args, **kwargs)	Add a recipe to the AnalysisPipeline.
`process_target`(target)	Process a node through the AnalysisPipeline.

ytree.analysis.analysis_pipeline.AnalysisPipeline.add_operation

`AnalysisPipeline.`**`add_operation`**(*function*, **args*, ***kwargs*)
Add an operation to the AnalysisPipeline.

An operation is a function that minimally takes in a target object and performs some actions on or with it. This function may alter the object's state, attach attributes, write out data, etc. Operations are used to create a pipeline of actions performed in sequence on a list of objects, such as all halos in a merger tree. The function can, optionally, return True or False to act as a filter, determining if the rest of the pipeline should be carried out (if True) or if the pipeline should stop and move on to the next object (if False).

Parameters

- **`function`** (`callable`) – The function to be called for each node/halo.
- **`*args`** (`positional arguments`) – Any additional positional arguments to be provided to the funciton.
- **`**kwargs`** (`keyword arguments`) – Any keyword arguments to be provided to the function.

ytree.analysis.analysis_pipeline.AnalysisPipeline.add_recipe

`AnalysisPipeline.`**`add_recipe`**(*function*, **args*, ***kwargs*)

Add a recipe to the AnalysisPipeline.

An recipe is a function that accepts an AnalysisPipeline and adds a series of operations with calls to add_operation. This is a way of creating a shortcut for a series of operations.

Parameters

- **`function`** (*`callable`*) – A function accepting an AnalysisPipeline object.
- **`*args`** (*`positional arguments`*) – Any additional positional arguments to be provided to the funciton.
- **`**kwargs`** (*`keyword arguments`*) – Any keyword arguments to be provided to the function.

Examples

```
>>> def print_field_value(node, field):
...     print (f"Node {node} has {field} of {node[field]}.")
>>>
>>> def print_many_things(pipeline, fields):
...    for field in fields:
...        pipeline.add_operation(print_field_value, field)
>>>
>>> ap = ytree.AnalysisPipeline()
>>> ap.add_recipe(print_many_things, ["mass", "virial_radius"])
```

ytree.analysis.analysis_pipeline.AnalysisPipeline.process_target

`AnalysisPipeline.`**`process_target`**(*target*)

Process a node through the AnalysisPipeline.

All operations added to the AnalysisPipeline will be run on the provided target.

Parameters **`target`** (*`TreeNode`*) – The node on which to run the analysis pipeline.

ytree.analysis.analysis_operators.AnalysisOperation

`class` `ytree.analysis.analysis_operators.`**`AnalysisOperation`**(*function*, **args*, ***kwargs*)

An analysis task performed by an AnalysisPipeline.

This is an internal class that facilitates keeping track of a function, arguments, and keyword arguments that together represent a single operation in a pipeline.

Parameters **`function`** (*`callable`*) – A function that minimally accepts a *`TreeNode`* object. The function may also accept additional positional and keyword arguments.

`__init__`(*function*, **args*, ***kwargs*)

Initialize self. See help(type(self)) for accurate signature.

Methods

`__init__`(function, *args, **kwargs)	Initialize self.

Parallelism

Functions for iterating over trees and/or nodes in parallel.

`parallel_trees`(trees[, save_every, . . .])	Iterate over a list of trees in parallel.
`parallel_tree_nodes`(tree[, group, njobs, . . .])	Iterate over nodes in a single tree in parallel.
`parallel_nodes`(trees[, group, save_every, . . .])	Iterate over all nodes in a list of trees in parallel.

ytree.utilities.parallel.parallel_trees

`ytree.utilities.parallel.`**`parallel_trees`**(*trees*, *save_every=None*, *filename=None*, *njobs=0*, *dynamic=False*)

Iterate over a list of trees in parallel.

Trees are divided up between the available processor groups. Analysis field values can then be assigned to halos within the tree. The trees will be saved either at the end of the loop or after a number of trees given by the `save_every` keyword are completed.

This uses the yt `parallel_objects` function, which is parallelized with MPI underneath and so is suitable for parallelism across compute nodes.

Parameters

- **`trees`** (list of `TreeNode` objects) – The trees to be iterated over in parallel.
- **`save_every`** (*`optional, int or False`*) – Number of trees to be completed before results are saved. This is used to save intermediate results in case scripts need to be restarted. If None, save will only occur after iterating over all trees. If False, no saving will be done. Default: None
- **`filename`** (*`optional, string`*) – The name of the new arbor to be saved. If None, the naming convention will follow the filename keyword of the `save_arbor` function. Default: None
- **`njobs`** (*`optional, int`*) – The number of process groups for parallel iteration. Set to 0 to make the same number of process groups as available processors. Hence, each tree will be allocated to a single processor. Set to a number less than the total number of processors to create groups with multiple processors, which will allow for further parallelization within a tree. For example, running with 8 processors and setting njobs to 4 will result in 4 groups of 2 processors each. Default: 0
- **`dynamic`** (*`optional, bool`*) – Set to False to divide iterations evenly among process groups. Set to True to allocate iterations with a task queue. If True, the number of processors available will be one fewer than the total as one will act as the task queue server. Default: False

Examples

```
>>> import ytree
>>> a = ytree.load("arbor/arbor.h5")
>>> a.add_analysis_field("test_field", default=-1, units="Msun")
>>> trees = list(a[:])
>>> for tree in ytree.parallel_trees(trees):
...     for node in tree["forest"]:
...         node["test_field"] = 2 * node["mass"] # some analysis
```

See also:

parallel_tree_nodes, *parallel_nodes*

ytree.utilities.parallel.parallel_tree_nodes

ytree.utilities.parallel.**parallel_tree_nodes**(*tree*, *group='forest'*, *njobs=0*, *dynamic=False*)

Iterate over nodes in a single tree in parallel.

Nodes are divided up between the available processor groups. Analysis field values can then be assigned to each node (halo).

Note, unlike the parallel_trees and parallel_nodes function, no saving is performed internally. Results saving with the *save_arbor* must be done manually.

This uses the yt parallel_objects function, which is parallelized with MPI underneath and so is suitable for parallelism across compute nodes.

Parameters

- **tree** (*TreeNode*) – The tree whose nodes will be iterated over.
- **group** (*optional, str ("forest", "tree", or "prog")*) – Determines the nodes to be iterated over in the tree: "forest" for all nodes in the forest, "tree" for all nodes in the tree, or "prog" for all nodes in the line of main progenitors. Default: "forest"
- **njobs** (*optional, int*) – The number of process groups for parallel iteration. Set to 0 to make the same number of process groups as available processors. Hence, each node will be allocated to a single processor. Set to a number less than the total number of processors to create groups with multiple processors, which will allow for further parallelization. For example, running with 8 processors and setting njobs to 4 will result in 4 groups of 2 processors each. Default: 0
- **dynamic** (*optional, bool*) – Set to False to divide iterations evenly among process groups. Set to True to allocate iterations with a task queue. If True, the number of processors available will be one fewer than the total as one will act as the task queue server. Default: False

Examples

```
>>> import ytree
>>> a = ytree.load("arbor/arbor.h5")
>>> a.add_analysis_field("test_field", default=-1, units="Msun")
>>> trees = list(a[:])
>>> for tree in trees:
...     for node in ytree.parallel_tree_nodes(tree):
...         node["test_field"] = 2 * node["mass"] # some analysis
```

See also:

parallel_trees, *parallel_nodes*

ytree.utilities.parallel.parallel_nodes

ytree.utilities.parallel.**parallel_nodes**(*trees*, *group='forest'*, *save_every=None*, *filename=None*, *njobs=None*, *dynamic=None*)

Iterate over all nodes in a list of trees in parallel.

Both trees and/or nodes within a tree are divided up between available process groups using multi-level parallelism. Analysis field values can then be assigned to all nodes (halos). Trees will be saved either at the end of the loop over all trees or after a number of trees given by the `save_every` keyword are completed.

This uses the yt `parallel_objects` function, which is parallelized with MPI underneath and so is suitable for parallelism across compute nodes.

Parameters

- **trees** (list of *TreeNode* objects) – The trees to be iterated over in parallel.
- **group** (*optional, str ("forest", "tree", or "prog")*) – Determines the nodes to be iterated over in the tree: "forest" for all nodes in the forest, "tree" for all nodes in the tree, or "prog" for all nodes in the line of main progenitors. Default: "forest"
- **save_every** (*optional, int or False*) – Number of trees to be completed before results are saved. This is used to save intermediate results in case scripts need to be restarted. If None, save will only occur after iterating over all trees. If False, no saving will be done. Default: None
- **filename** (*optional, string*) – The name of the new arbor to be saved. If None, the naming convention will follow the filename keyword of the *save_arbor* function. Default: None
- **njobs** (*optional, tuple of ints*) – The number of process groups for parallel iteration over trees and nodes within each tree. The first value sets behavior for iteration over trees and the second for iteration over nodes in a tree. For example, set to (1, 0) to parallelize only over nodes in a tree and (0, 1) to parallelize only over trees. For multi-level parallelism set the first value to a number less than the total number of processors and the second to 0. For example, if running with 8 processors, set njobs to (2, 0) to iterate over each tree with a group of 4 processors. Within each tree, each of the 4 processors in the group will work on a single node. If set to None, njobs will be set to (0, 1) if there are most trees than processors (tree parallel) and (1, 0) otherwise (node parallel). Default: None
- **dynamic** (*optional, tuples of bools*) – Toggles task queue on/off for parallelism over trees (first value) and nodes within a tree (second). Set to a value False to divide iterations evenly among process groups. Set to True to allocate iterations with a task queue. If True, the number of processors available will be one fewer than the total as one will act as the task queue server. Yes, this can be set to (True, True). Try it. Default: (False, False)

Examples

```
>>> import ytree
>>> a = ytree.load("arbor/arbor.h5")
>>> a.add_analysis_field("test_field", default=-1, units="Msun")
>>> trees = list(a[:])
```

(continues on next page)

(continued from previous page)

```
>>> for node in ytree.parallel_nodes(trees):
...     node["test_field"] = 2 * node["mass"] # some analysis
```

See also:

parallel_trees, *parallel_tree_nodes*

Internal Classes

Base Classes

All frontends inherit from these base classes for arbor, fields, and i/o.

Arbor(filename)	Base class for all Arbor classes.
SegmentedArbor(filename)	Arbor subclass for multi-file datasets where an entire merger tree is contained within a file (i.e., no overlap).
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.
Detector	Base class for detecting field dependencies and testing operations.
FieldDetector(arbor[, name])	A fake field data container used to calculate dependencies.
SelectionDetector(arbor)	A TreeNode-like object to test select_halos criteria.
FieldInfoContainer(arbor)	A container for information about fields.
FieldContainer(arbor)	A container for field data.
FieldIO(arbor[, default_dtype])	Base class for FieldIO classes.
TreeFieldIO(arbor[, default_dtype])	IO class for getting fields for a tree.
DefaultRootFieldIO(arbor[, default_dtype])	Class for getting root fields from arbors that have no specialized storage for root fields.
DataFile(filename)	Base class for data files.
CatalogDataFile(filename, arbor)	Base class for halo catalog files.

ytree.data_structures.arbor.SegmentedArbor

class ytree.data_structures.arbor.**SegmentedArbor**(*filename*)

Arbor subclass for multi-file datasets where an entire merger tree is contained within a file (i.e., no overlap). This permits the definition of a useful _node_io_loop_prepare function.

__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.

Continued on next page

Table 15 – continued from previous page

`add_analysis_field`(name, units[, dtype, default])	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.
`add_vector_field`(name)	Add vector fields for a set of x,y,z component fields.
`get_nodes_from_selection`(*args, **kwargs)	
`get_yt_selection`(*args, **kwargs)	
`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.
`reset_node`(tree_node)	Reset all data structures for a single node.
`save_arbor`(**kwargs)	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.

Attributes

`arr`	Create a unyt_array using the Arbor's unit registry.
`box_size`	The simulation box size.
`field_info`	A dictionary containing information for each available field.
`hubble_constant`	Value of the Hubble parameter.
`is_planted`	Determine if trees have been planted.
`omega_lambda`	
`omega_matter`	
`omega_radiation`	
`quan`	Create a unyt_quantity using the Arbor's unit registry.
`size`	Return total number of trees.
`unit_registry`	Unit system registry.
`ytds`	

ytree.data_structures.arbor.CatalogArbor

class `ytree.data_structures.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.

Unlike formats where tree information is stored in single file, halos are scattered about multiple catalog files. This requires us to store the root TreeNode objects and their full assemblies.

`__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[, dtype, default])	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.
`add_vector_field`(name)	Add vector fields for a set of x,y,z component fields.
`get_nodes_from_selection`(*args, **kwargs)	
`get_yt_selection`(*args, **kwargs)	
`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.
`reset_node`(tree_node)	Reset all data structures for a single node.
`save_arbor`(**kwargs)	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.

Attributes

`arr`	Create a unyt_array using the Arbor's unit registry.
`box_size`	The simulation box size.
`field_info`	A dictionary containing information for each available field.
`hubble_constant`	Value of the Hubble parameter.
`is_planted`	Determine if trees have been planted.
`omega_lambda`	
`omega_matter`	
`omega_radiation`	
`quan`	Create a unyt_quantity using the Arbor's unit registry.
`size`	Return total number of trees.
`unit_registry`	Unit system registry.
`ytds`	

ytree.data_structures.detection.Detector

class ytree.data_structures.detection.**Detector**

Base class for detecting field dependencies and testing operations.

`__init__`()

Initialize self. See help(type(self)) for accurate signature.

Methods

clear()	
copy()	
fromkeys	Create a new dictionary with keys from iterable and values set to value.
get	Return the value for key if key is in the dictionary, else default.
items()	
keys()	
pop(k[,d])	If key is not found, d is returned if given, otherwise KeyError is raised
popitem()	2-tuple; but raise KeyError if D is empty.
setdefault	Insert key with a value of default if key is not in the dictionary.
update([E,]**F)	If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]
values()	

Attributes

default_factory	Factory for default value called by __missing__().

ytree.data_structures.detection.FieldDetector

class ytree.data_structures.detection.**FieldDetector**(*arbor*, *name=None*)
A fake field data container used to calculate dependencies.

__init__(*arbor*, *name=None*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(arbor[, name])	Initialize self.
clear()	
copy()	
fromkeys	Create a new dictionary with keys from iterable and values set to value.
get	Return the value for key if key is in the dictionary, else default.
items()	
keys()	
pop(k[,d])	If key is not found, d is returned if given, otherwise KeyError is raised
popitem()	2-tuple; but raise KeyError if D is empty.

Continued on next page

Table 21 – continued from previous page

`setdefault`	Insert key with a value of default if key is not in the dictionary.
`update([E, ]**F)`	If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]
`values()`	

Attributes

`default_factory`	Factory for default value called by __missing__().

ytree.data_structures.detection.SelectionDetector

class `ytree.data_structures.detection.`**`SelectionDetector`**(*arbor*)

A TreeNode-like object to test select_halos criteria.

`__init__`(*arbor*)

Initialize self. See help(type(self)) for accurate signature.

Methods

`__init__(arbor)`	Initialize self.
`clear()`	
`copy()`	
`fromkeys`	Create a new dictionary with keys from iterable and values set to value.
`get`	Return the value for key if key is in the dictionary, else default.
`items()`	
`keys()`	
`pop(k[,d])`	If key is not found, d is returned if given, otherwise KeyError is raised
`popitem()`	2-tuple; but raise KeyError if D is empty.
`setdefault`	Insert key with a value of default if key is not in the dictionary.
`update([E, ]**F)`	If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]
`values()`	

Attributes

`default_factory`	Factory for default value called by __missing__().

ytree.data_structures.fields.FieldInfoContainer

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

A container for information about fields.

__init__(*arbor*)

Initialize self. See help(type(self)) for accurate signature.

Methods

`__init__`(arbor)	Initialize self.
`add_alias_field`(alias, field[, units, force_add])	Add an alias field.
`add_analysis_field`(name, units[, dtype, default])	Add an analysis field.
`add_derived_field`(name, function[, units, . . .])	Add a derived field.
`add_vector_field`(fieldname)	Add vector and magnitude fields for a field with x/y/z components.
`clear`()	
`copy`()	
`fromkeys`	Create a new dictionary with keys from iterable and values set to value.
`get`	Return the value for key if key is in the dictionary, else default.
`items`()	
`keys`()	
`pop`(k[,d])	If key is not found, d is returned if given, otherwise KeyError is raised
`popitem`()	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`	Insert key with a value of default if key is not in the dictionary.
`setup_aliases`()	Add aliases defined in the alias_fields tuple for each frontend.
`setup_derived_fields`()	Add stock derived fields.
`setup_known_fields`()	Add units for fields on disk as defined in the known_fields tuple.
`setup_vector_fields`()	Add vector and magnitude fields.
`update`([E,]**F)	If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]
`values`()	

Attributes

alias_fields
data_types
known_fields
vector_fields

ytree.data_structures.fields.FieldContainer

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

A container for field data.

__init__(*arbor*)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(arbor)	Initialize self.
clear()	
copy()	
fromkeys	Create a new dictionary with keys from iterable and values set to value.
get	Return the value for key if key is in the dictionary, else default.
items()	
keys()	
pop(k[,d])	If key is not found, d is returned if given, otherwise KeyError is raised
popitem()	2-tuple; but raise KeyError if D is empty.
setdefault	Insert key with a value of default if key is not in the dictionary.
update([E,]**F)	If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]
values()	

ytree.data_structures.io.FieldIO

class ytree.data_structures.io.**FieldIO**(*arbor*, *default_dtype=<class 'numpy.float64'>*)

Base class for FieldIO classes.

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

__init__(*arbor*, *default_dtype=<class 'numpy.float64'>*)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(arbor[, default_dtype])	Initialize self.

Continued on next page

Table 28 – continued from previous page

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

ytree.data_structures.io.TreeFieldIO

class `ytree.data_structures.io.`**`TreeFieldIO`**(*arbor*, *default_dtype=<class 'numpy.float64'>*)

IO class for getting fields for a tree.

`__init__`(*arbor*, *default_dtype=<class 'numpy.float64'>*)

Initialize self. See help(type(self)) for accurate signature.

Methods

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

ytree.data_structures.io.DefaultRootFieldIO

class `ytree.data_structures.io.`**`DefaultRootFieldIO`**(*arbor*, *default_dtype=<class 'numpy.float64'>*)

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

`__init__`(*arbor*, *default_dtype=<class 'numpy.float64'>*)

Initialize self. See help(type(self)) for accurate signature.

Methods

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

ytree.data_structures.io.DataFile

class `ytree.data_structures.io.`**`DataFile`**(*filename*)

Base class for data files.

This class allows us keep files open during i/o heavy operations and to keep things like caches of fields.

`__init__`(*filename*)

Initialize self. See help(type(self)) for accurate signature.

Methods

`__init__`(filename)	Initialize self.
`close`()	
`open`()	

ytree.data_structures.io.CatalogDataFile

class ytree.data_structures.io.**CatalogDataFile**(*filename*, *arbor*)
Base class for halo catalog files.

__init__(*filename*, *arbor*)
Initialize self. See help(type(self)) for accurate signature.

Methods

`__init__`(filename, arbor)	Initialize self.
`close`()	
`open`()	

Arbor Subclasses

Arbor subclasses for each frontend.

`AHFArbor`(filename[, log_filename, . . .])	Arbor for Amiga Halo Finder data.
`ConsistentTreesArbor`(filename)	Arbors loaded from consistent-trees tree_*.dat files.
`ConsistentTreesGroupArbor`(filename)	Arbors loaded from consistent-trees locations.dat files.
`ConsistentTreesHlistArbor`(filename)	Class for Arbors created from consistent-trees hlist_*.list files.
`ConsistentTreesHDF5Arbor`(filename[, access])	Arbors loaded from consistent-trees data converted into HDF5.
`LHaloTreeArbor`(*args, **kwargs)	Arbors for LHaloTree data.
`LHaloTreeHDF5Arbor`(filename[, . . .])	Arbors loaded from consistent-trees data converted into HDF5.
`MoriaArbor`(filename)	Arbors from Moria merger trees.
`RockstarArbor`(filename)	Class for Arbors created from Rockstar out_*.list files.
`TreeFarmArbor`(filename)	Class for Arbors created with `TreeFarm`.
`YTreeArbor`(filename)	Class for Arbors created from the `save_arbor` or `save_tree` functions.

ytree.frontends.ahf.arbor.AHFArbor

class ytree.frontends.ahf.arbor.**AHFArbor**(*filename*, *log_filename=None*, *hubble_constant=1.0*, *box_size=None*, *omega_matter=None*, *omega_lambda=None*)
Arbor for Amiga Halo Finder data.

__init__(*filename*, *log_filename=None*, *hubble_constant=1.0*, *box_size=None*, *omega_matter=None*, *omega_lambda=None*)
Initialize an Arbor given an input file.

Methods

`__init__`(filename[, log_filename, . . .])	Initialize an Arbor given an input file.

Continued on next page

Table 34 – continued from previous page

`add_alias_field`(alias, field[, units, force_add])	Add a field as an alias to another field.
`add_analysis_field`(name, units[, dtype, default])	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.
`add_vector_field`(name)	Add vector fields for a set of x,y,z component fields.
`get_nodes_from_selection`(*args, **kwargs)	
`get_yt_selection`(*args, **kwargs)	
`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.
`reset_node`(tree_node)	Reset all data structures for a single node.
`save_arbor`(**kwargs)	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.

Attributes

`arr`	Create a unyt_array using the Arbor's unit registry.
`box_size`	The simulation box size.
`field_info`	A dictionary containing information for each available field.
`hubble_constant`	Value of the Hubble parameter.
`is_planted`	Determine if trees have been planted.
`omega_lambda`	
`omega_matter`	
`omega_radiation`	
`quan`	Create a unyt_quantity using the Arbor's unit registry.
`size`	Return total number of trees.
`unit_registry`	Unit system registry.
`ytds`	

ytree.frontends.consistent_trees.arbor.ConsistentTreesArbor

class `ytree.frontends.consistent_trees.arbor.`**`ConsistentTreesArbor`**(*filename*)

Arbors loaded from consistent-trees tree_*.dat 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[, dtype, default])	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.
`add_vector_field`(name)	Add vector fields for a set of x,y,z component fields.
`get_nodes_from_selection`(*args, **kwargs)	
`get_yt_selection`(*args, **kwargs)	
`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.
`reset_node`(tree_node)	Reset all data structures for a single node.
`save_arbor`(**kwargs)	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.

Attributes

`arr`	Create a unyt_array using the Arbor's unit registry.
`box_size`	The simulation box size.
`field_info`	A dictionary containing information for each available field.
`hubble_constant`	Value of the Hubble parameter.
`is_planted`	Determine if trees have been planted.
`omega_lambda`	
`omega_matter`	
`omega_radiation`	
`quan`	Create a unyt_quantity using the Arbor's unit registry.
`size`	Return total number of trees.
`unit_registry`	Unit system registry.
`ytds`	

ytree.frontends.consistent_trees.arbor.ConsistentTreesGroupArbor

class `ytree.frontends.consistent_trees.arbor.`**`ConsistentTreesGroupArbor`**(*filename*)

Arbors loaded from consistent-trees locations.dat 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[, dtype, default])	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.
`add_vector_field`(name)	Add vector fields for a set of x,y,z component fields.
`get_nodes_from_selection`(*args, **kwargs)	
`get_yt_selection`(*args, **kwargs)	
`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.
`reset_node`(tree_node)	Reset all data structures for a single node.
`save_arbor`(**kwargs)	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.

Attributes

`arr`	Create a unyt_array using the Arbor's unit registry.
`box_size`	The simulation box size.
`field_info`	A dictionary containing information for each available field.
`hubble_constant`	Value of the Hubble parameter.
`is_planted`	Determine if trees have been planted.
`omega_lambda`	
`omega_matter`	
`omega_radiation`	
`quan`	Create a unyt_quantity using the Arbor's unit registry.
`size`	Return total number of trees.
`unit_registry`	Unit system registry.
`ytds`	

ytree.frontends.consistent_trees.arbor.ConsistentTreesHlistArbor

class `ytree.frontends.consistent_trees.arbor.`**`ConsistentTreesHlistArbor`**(*filename*)

Class for Arbors created from consistent-trees hlist_*.list files.

This is a hybrid type with multiple catalog files like the rockstar frontend, but with headers structured like consistent-trees.

`__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[, dtype, default])	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.
`add_vector_field`(name)	Add vector fields for a set of x,y,z component fields.
`get_nodes_from_selection`(*args, **kwargs)	
`get_yt_selection`(*args, **kwargs)	
`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.
`reset_node`(tree_node)	Reset all data structures for a single node.
`save_arbor`(**kwargs)	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.

Attributes

`arr`	Create a unyt_array using the Arbor's unit registry.
`box_size`	The simulation box size.
`field_info`	A dictionary containing information for each available field.
`hubble_constant`	Value of the Hubble parameter.
`is_planted`	Determine if trees have been planted.
`omega_lambda`	
`omega_matter`	
`omega_radiation`	
`quan`	Create a unyt_quantity using the Arbor's unit registry.
`size`	Return total number of trees.
`unit_registry`	Unit system registry.
`ytds`	

ytree.frontends.consistent_trees_hdf5.arbor.ConsistentTreesHDF5Arbor

class `ytree.frontends.consistent_trees_hdf5.arbor.`**`ConsistentTreesHDF5Arbor`**(*filename*, *access='tree'*)

Arbors loaded from consistent-trees data converted into HDF5.

`__init__`(*filename*, *access='tree'*)

Initialize an Arbor given an input file.

Methods

`__init__`(filename[, access])	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[, dtype, default])	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.
`add_vector_field`(name)	Add vector fields for a set of x,y,z component fields.
`get_nodes_from_selection`(*args, **kwargs)	
`get_yt_selection`(*args, **kwargs)	
`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.
`reset_node`(tree_node)	Reset all data structures for a single node.
`save_arbor`(**kwargs)	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.

Attributes

`arr`	Create a unyt_array using the Arbor's unit registry.
`box_size`	The simulation box size.
`field_info`	A dictionary containing information for each available field.
`hubble_constant`	Value of the Hubble parameter.
`is_planted`	Determine if trees have been planted.
`omega_lambda`	
`omega_matter`	
`omega_radiation`	
`quan`	Create a unyt_quantity using the Arbor's unit registry.
`size`	Return total number of trees.
`unit_registry`	Unit system registry.
`ytds`	

ytree.frontends.lhalotree.arbor.LHaloTreeArbor

class `ytree.frontends.lhalotree.arbor.`**`LHaloTreeArbor`**(**args*, ***kwargs*)

Arbors for LHaloTree data.

`__init__`(**args*, ***kwargs*)

Added reader class to allow fast access of header info.

Methods

`__init__`(*args, **kwargs)	Added reader class to allow fast access of header info.
`add_alias_field`(alias, field[, units, force_add])	Add a field as an alias to another field.
`add_analysis_field`(name, units[, dtype, default])	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.
`add_vector_field`(name)	Add vector fields for a set of x,y,z component fields.
`get_nodes_from_selection`(*args, **kwargs)	
`get_yt_selection`(*args, **kwargs)	
`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.
`reset_node`(tree_node)	Reset all data structures for a single node.
`save_arbor`(**kwargs)	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.

Attributes

`arr`	Create a unyt_array using the Arbor's unit registry.
`box_size`	The simulation box size.
`field_info`	A dictionary containing information for each available field.
`hubble_constant`	Value of the Hubble parameter.
`is_planted`	Determine if trees have been planted.
`omega_lambda`	
`omega_matter`	
`omega_radiation`	
`quan`	Create a unyt_quantity using the Arbor's unit registry.
`size`	Return total number of trees.
`unit_registry`	Unit system registry.
`ytds`	

ytree.frontends.lhalotree_hdf5.arbor.LHaloTreeHDF5Arbor

class `ytree.frontends.lhalotree_hdf5.arbor.`**`LHaloTreeHDF5Arbor`**(*filename*, *hubble_constant=1.0*, *box_size=None*, *omega_matter=None*, *omega_lambda=None*)

Arbors loaded from consistent-trees data converted into HDF5.

__init__ (*filename*, *hubble_constant=1.0*, *box_size=None*, *omega_matter=None*, *omega_lambda=None*)
Initialize an Arbor given an input file.

Methods

`__init__`(filename[, hubble_constant, . . .])	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[, dtype, default])	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.
`add_vector_field`(name)	Add vector fields for a set of x,y,z component fields.
`get_nodes_from_selection`(*args, **kwargs)	
`get_yt_selection`(*args, **kwargs)	
`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.
`reset_node`(tree_node)	Reset all data structures for a single node.
`save_arbor`(**kwargs)	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.

Attributes

`arr`	Create a unyt_array using the Arbor's unit registry.
`box_size`	The simulation box size.
`field_info`	A dictionary containing information for each available field.
`hubble_constant`	Value of the Hubble parameter.
`is_planted`	Determine if trees have been planted.
`omega_lambda`	
`omega_matter`	
`omega_radiation`	
`quan`	Create a unyt_quantity using the Arbor's unit registry.
`size`	Return total number of trees.
`unit_registry`	Unit system registry.
`ytds`	

ytree.frontends.moria.arbor.MoriaArbor

class `ytree.frontends.moria.arbor.`**`MoriaArbor`**(*filename*)
Arbors from Moria merger trees.

`__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[, dtype, default])	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.
`add_vector_field`(name)	Add vector fields for a set of x,y,z component fields.
`get_nodes_from_selection`(*args, **kwargs)	
`get_yt_selection`(*args, **kwargs)	
`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.
`reset_node`(tree_node)	Reset all data structures for a single node.
`save_arbor`(**kwargs)	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.

Attributes

`arr`	Create a unyt_array using the Arbor's unit registry.
`box_size`	The simulation box size.
`field_info`	A dictionary containing information for each available field.
`hubble_constant`	Value of the Hubble parameter.
`is_planted`	Determine if trees have been planted.
`omega_lambda`	
`omega_matter`	
`omega_radiation`	
`quan`	Create a unyt_quantity using the Arbor's unit registry.
`size`	Return total number of trees.
`unit_registry`	Unit system registry.
`ytds`	

ytree.frontends.rockstar.arbor.RockstarArbor

class `ytree.frontends.rockstar.arbor.`**`RockstarArbor`**(*filename*)
Class for Arbors created from Rockstar out_*.list files. Use only descendent IDs to determine tree relationship.

`__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[, dtype, default])	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.
`add_vector_field`(name)	Add vector fields for a set of x,y,z component fields.
`get_nodes_from_selection`(*args, **kwargs)	
`get_yt_selection`(*args, **kwargs)	
`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.
`reset_node`(tree_node)	Reset all data structures for a single node.
`save_arbor`(**kwargs)	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.

Attributes

`arr`	Create a unyt_array using the Arbor's unit registry.
`box_size`	The simulation box size.
`field_info`	A dictionary containing information for each available field.
`hubble_constant`	Value of the Hubble parameter.
`is_planted`	Determine if trees have been planted.
`omega_lambda`	
`omega_matter`	
`omega_radiation`	
`quan`	Create a unyt_quantity using the Arbor's unit registry.
`size`	Return total number of trees.
`unit_registry`	Unit system registry.
`ytds`	

ytree.frontends.treefarm.arbor.TreeFarmArbor

class `ytree.frontends.treefarm.arbor.`**`TreeFarmArbor`**(*filename*)
Class for Arbors created with `TreeFarm`.

__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[, dtype, default])	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.
`add_vector_field`(name)	Add vector fields for a set of x,y,z component fields.
`get_nodes_from_selection`(*args, **kwargs)	
`get_yt_selection`(*args, **kwargs)	
`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.
`reset_node`(tree_node)	Reset all data structures for a single node.
`save_arbor`(**kwargs)	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.

Attributes

`arr`	Create a unyt_array using the Arbor's unit registry.
`box_size`	The simulation box size.
`field_info`	A dictionary containing information for each available field.
`hubble_constant`	Value of the Hubble parameter.
`is_planted`	Determine if trees have been planted.
`omega_lambda`	
`omega_matter`	
`omega_radiation`	
`quan`	Create a unyt_quantity using the Arbor's unit registry.
`size`	Return total number of trees.
`unit_registry`	Unit system registry.
`ytds`	

ytree.frontends.ytree.arbor.YTreeArbor

class `ytree.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[, dtype, default])	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.
`add_vector_field`(name)	Add vector fields for a set of x,y,z component fields.
`get_nodes_from_selection`(container)	Generate TreeNodes from a yt data container.
`get_yt_selection`([above, below, equal, . . .])	Get a selection of halos meeting given criteria.
`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.
`reset_node`(tree_node)	Reset all data structures for a single node.
`save_arbor`(**kwargs)	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.

Attributes

`arr`	Create a unyt_array using the Arbor's unit registry.
`box_size`	The simulation box size.
`field_info`	A dictionary containing information for each available field.
`hubble_constant`	Value of the Hubble parameter.
`is_planted`	Determine if trees have been planted.
`omega_lambda`	
`omega_matter`	
`omega_radiation`	
`quan`	Create a unyt_quantity using the Arbor's unit registry.
`size`	Return total number of trees.
`unit_registry`	Unit system registry.
`ytds`	Load as a yt dataset.

FieldInfo Subclasses

Subclasses for frontend-specific field definitions.

AHFFieldInfo(arbor)
ConsistentTreesFieldInfo(arbor)
ConsistentTreesHDF5FieldInfo(arbor)
LHaloTreeFieldInfo(arbor)
LHaloTreeHDF5FieldInfo(arbor)
MoriaFieldInfo(arbor)
RockstarFieldInfo(arbor)
TreeFarmFieldInfo(arbor)

ytree.frontends.ahf.fields.AHFFieldInfo

class ytree.frontends.ahf.fields.**AHFFieldInfo**(*arbor*)

__init__(*arbor*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(arbor)	Initialize self.
add_alias_field(alias, field[, units, force_add])	Add an alias field.
add_analysis_field(name, units[, dtype, default])	Add an analysis field.
add_derived_field(name, function[, units, . . .])	Add a derived field.
add_vector_field(fieldname)	Add vector and magnitude fields for a field with x/y/z components.
clear()	
copy()	
fromkeys	Create a new dictionary with keys from iterable and values set to value.
get	Return the value for key if key is in the dictionary, else default.
items()	
keys()	
pop(k[,d])	If key is not found, d is returned if given, otherwise KeyError is raised
popitem()	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	Insert key with a value of default if key is not in the dictionary.
setup_aliases()	Add aliases defined in the alias_fields tuple for each frontend.
setup_derived_fields()	Add stock derived fields.
setup_known_fields()	Add units for fields on disk as defined in the known_fields tuple.
setup_vector_fields()	Add vector and magnitude fields.

Continued on next page

Table 57 – continued from previous page

update([E,]**F)	If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]
values()	

Attributes

alias_fields	
data_types	
known_fields	
vector_fields	

ytree.frontends.consistent_trees.fields.ConsistentTreesFieldInfo

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

__init__(*arbor*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(arbor)	Initialize self.
add_alias_field(alias, field[, units, force_add])	Add an alias field.
add_analysis_field(name, units[, dtype, default])	Add an analysis field.
add_derived_field(name, function[, units, . . .])	Add a derived field.
add_vector_field(fieldname)	Add vector and magnitude fields for a field with x/y/z components.
clear()	
copy()	
fromkeys	Create a new dictionary with keys from iterable and values set to value.
get	Return the value for key if key is in the dictionary, else default.
items()	
keys()	
pop(k[,d])	If key is not found, d is returned if given, otherwise KeyError is raised
popitem()	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	Insert key with a value of default if key is not in the dictionary.

Continued on next page

Table 59 – continued from previous page

setup_aliases()	Add aliases defined in the alias_fields tuple for each frontend.
setup_derived_fields()	Add stock derived fields.
setup_known_fields()	Add units for fields on disk as defined in the known_fields tuple.
setup_vector_fields()	Add vector and magnitude fields.
update([E,]**F)	If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]
values()	

Attributes

alias_fields	
data_types	
known_fields	
vector_fields	

ytree.frontends.consistent_trees_hdf5.fields.ConsistentTreesHDF5FieldInfo

class ytree.frontends.consistent_trees_hdf5.fields.**ConsistentTreesHDF5FieldInfo**(*arbor*)

__init__(*arbor*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(arbor)	Initialize self.
add_alias_field(alias, field[, units, force_add])	Add an alias field.
add_analysis_field(name, units[, dtype, default])	Add an analysis field.
add_derived_field(name, function[, units, ...])	Add a derived field.
add_vector_field(fieldname)	Add vector and magnitude fields for a field with x/y/z components.
clear()	
copy()	
fromkeys	Create a new dictionary with keys from iterable and values set to value.
get	Return the value for key if key is in the dictionary, else default.
items()	
keys()	
pop(k[,d])	If key is not found, d is returned if given, otherwise KeyError is raised

Continued on next page

Table 61 – continued from previous page

`popitem`()	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`	Insert key with a value of default if key is not in the dictionary.
`setup_aliases`()	Add aliases defined in the alias_fields tuple for each frontend.
`setup_derived_fields`()	Add stock derived fields.
`setup_known_fields`()	Add units for fields on disk as defined in the known_fields tuple.
`setup_vector_fields`()	Add vector and magnitude fields.
`update`([E,]**F)	If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]
`values`()	

Attributes

`alias_fields`
`data_types`
`known_fields`
`vector_fields`

ytree.frontends.lhalotree.fields.LHaloTreeFieldInfo

class `ytree.frontends.lhalotree.fields.`**`LHaloTreeFieldInfo`**(*arbor*)

`__init__`(*arbor*)
Initialize self. See help(type(self)) for accurate signature.

Methods

`__init__`(arbor)	Initialize self.
`add_alias_field`(alias, field[, units, force_add])	Add an alias field.
`add_analysis_field`(name, units[, dtype, default])	Add an analysis field.
`add_derived_field`(name, function[, units, ...])	Add a derived field.
`add_vector_field`(fieldname)	Add vector and magnitude fields for a field with x/y/z components.
`clear`()	
`copy`()	
`fromkeys`	Create a new dictionary with keys from iterable and values set to value.

Continued on next page

Table 63 – continued from previous page

`get`	Return the value for key if key is in the dictionary, else default.
`items()`	
`keys()`	
`pop(k[,d])`	If key is not found, d is returned if given, otherwise KeyError is raised
`popitem()`	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`	Insert key with a value of default if key is not in the dictionary.
`setup_aliases()`	Add aliases defined in the alias_fields tuple for each frontend.
`setup_derived_fields()`	Add stock derived fields.
`setup_known_fields()`	Add units for fields on disk as defined in the known_fields tuple.
`setup_vector_fields()`	Add vector and magnitude fields.
`update([E, ]**F)`	If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]
`values()`	

Attributes

`alias_fields`
`data_types`
`known_fields`
`vector_fields`

ytree.frontends.lhalotree_hdf5.fields.LHaloTreeHDF5FieldInfo

class ytree.frontends.lhalotree_hdf5.fields.**LHaloTreeHDF5FieldInfo**(*arbor*)

__init__(*arbor*)
Initialize self. See help(type(self)) for accurate signature.

Methods

`__init__`(arbor)	Initialize self.
`add_alias_field(alias, field[, units, force_add])`	Add an alias field.
`add_analysis_field(name, units[, dtype, default])`	Add an analysis field.
`add_derived_field(name, function[, units, ...])`	Add a derived field.

Continued on next page

Table 65 – continued from previous page

`add_vector_field`(fieldname)	Add vector and magnitude fields for a field with x/y/z components.
`clear`()	
`copy`()	
`fromkeys`	Create a new dictionary with keys from iterable and values set to value.
`get`	Return the value for key if key is in the dictionary, else default.
`items`()	
`keys`()	
`pop`(k[,d])	If key is not found, d is returned if given, otherwise KeyError is raised
`popitem`()	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`	Insert key with a value of default if key is not in the dictionary.
`setup_aliases`()	Add aliases defined in the alias_fields tuple for each frontend.
`setup_derived_fields`()	Add stock derived fields.
`setup_known_fields`()	Add units for all <fieldname>_<number> fields as well.
`setup_vector_fields`()	Add vector and magnitude fields.
`update`([E,]**F)	If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]
`values`()	

Attributes

`alias_fields`
`data_types`
`known_fields`
`vector_fields`

ytree.frontends.moria.fields.MoriaFieldInfo

class `ytree.frontends.moria.fields.`**`MoriaFieldInfo`**(*arbor*)

`__init__`(*arbor*)
Initialize self. See help(type(self)) for accurate signature.

Methods

`__init__`(arbor)	Initialize self.

Continued on next page

Table 67 – continued from previous page

add_alias_field(alias, field[, units, force_add])	Add an alias field.
add_analysis_field(name, units[, dtype, default])	Add an analysis field.
add_derived_field(name, function[, units, . . .])	Add a derived field.
add_vector_field(fieldname)	Add vector and magnitude fields for a field with x/y/z components.
clear()	
copy()	
fromkeys	Create a new dictionary with keys from iterable and values set to value.
get	Return the value for key if key is in the dictionary, else default.
items()	
keys()	
pop(k[,d])	If key is not found, d is returned if given, otherwise KeyError is raised
popitem()	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	Insert key with a value of default if key is not in the dictionary.
setup_aliases()	Add aliases defined in the alias_fields tuple for each frontend.
setup_derived_fields()	Add stock derived fields.
setup_known_fields()	Add units for all <fieldname>_<number> fields as well.
setup_vector_fields()	Add vector and magnitude fields.
update([E,]**F)	If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]
values()	

Attributes

alias_fields
data_types
known_fields
vector_fields

ytree.frontends.rockstar.fields.RockstarFieldInfo

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

__init__(*arbor*)
Initialize self. See help(type(self)) for accurate signature.

Methods

`__init__`(arbor)	Initialize self.
`add_alias_field`(alias, field[, units, force_add])	Add an alias field.
`add_analysis_field`(name, units[, dtype, default])	Add an analysis field.
`add_derived_field`(name, function[, units, . . .])	Add a derived field.
`add_vector_field`(fieldname)	Add vector and magnitude fields for a field with x/y/z components.
`clear`()	
`copy`()	
`fromkeys`	Create a new dictionary with keys from iterable and values set to value.
`get`	Return the value for key if key is in the dictionary, else default.
`items`()	
`keys`()	
`pop`(k[,d])	If key is not found, d is returned if given, otherwise KeyError is raised
`popitem`()	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`	Insert key with a value of default if key is not in the dictionary.
`setup_aliases`()	Add aliases defined in the alias_fields tuple for each frontend.
`setup_derived_fields`()	Add stock derived fields.
`setup_known_fields`()	Add units for fields on disk as defined in the known_fields tuple.
`setup_vector_fields`()	Add vector and magnitude fields.
`update`([E,]**F)	If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]
`values`()	

Attributes

`alias_fields`
`data_types`
`known_fields`
`vector_fields`

ytree.frontends.treefarm.fields.TreeFarmFieldInfo

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

__init__(*arbor*)
Initialize self. See help(type(self)) for accurate signature.

Methods

`__init__`(arbor)	Initialize self.
`add_alias_field`(alias, field[, units, force_add])	Add an alias field.
`add_analysis_field`(name, units[, dtype, default])	Add an analysis field.
`add_derived_field`(name, function[, units, . . .])	Add a derived field.
`add_vector_field`(fieldname)	Add vector and magnitude fields for a field with x/y/z components.
`clear`()	
`copy`()	
`fromkeys`	Create a new dictionary with keys from iterable and values set to value.
`get`	Return the value for key if key is in the dictionary, else default.
`items`()	
`keys`()	
`pop`(k[,d])	If key is not found, d is returned if given, otherwise KeyError is raised
`popitem`()	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`	Insert key with a value of default if key is not in the dictionary.
`setup_aliases`()	Add aliases defined in the alias_fields tuple for each frontend.
`setup_derived_fields`()	Add stock derived fields.
`setup_known_fields`()	Add units for fields on disk as defined in the known_fields tuple.
`setup_vector_fields`()	Add vector and magnitude fields.
`update`([E,]**F)	If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]
`values`()	

Attributes

`alias_fields`
`data_types`
`known_fields`
`vector_fields`

FieldIO Subclasses

Subclasses for data i/o from a whole dataset.

ConsistentTreesTreeFieldIO(arbor[, …])	
ConsistentTreesHDF5TreeFieldIO(arbor[, …])	
ConsistentTreesHDF5RootFieldIO(arbor[, …])	Read in fields for first node in all trees/forest.
LHaloTreeTreeFieldIO(arbor[, default_dtype])	
LHaloTreeRootFieldIO(arbor[, default_dtype])	
LHaloTreeHDF5TreeFieldIO(arbor[, default_dtype])	
MoriaTreeFieldIO(arbor[, default_dtype])	
YTreeTreeFieldIO(arbor[, default_dtype])	
YTreeRootFieldIO(arbor[, default_dtype])	

ytree.frontends.consistent_trees.io.ConsistentTreesTreeFieldIO

class ytree.frontends.consistent_trees.io.**ConsistentTreesTreeFieldIO**(*arbor*, *default_dtype=<class 'numpy.float64'>*)

__init__(*arbor*, *default_dtype=<class 'numpy.float64'>*)
Initialize self. See help(type(self)) for accurate signature.

Methods

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

ytree.frontends.consistent_trees_hdf5.io.ConsistentTreesHDF5TreeFieldIO

class ytree.frontends.consistent_trees_hdf5.io.**ConsistentTreesHDF5TreeFieldIO**(*arbor*, *default_dtype=<clas 'numpy.float64'>*)

__init__(*arbor*, *default_dtype=<class 'numpy.float64'>*)
Initialize self. See help(type(self)) for accurate signature.

Methods

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

ytree.frontends.consistent_trees_hdf5.io.ConsistentTreesHDF5RootFieldIO

class ytree.frontends.consistent_trees_hdf5.io.**ConsistentTreesHDF5RootFieldIO**(*arbor*, *de-fault_dtype=<clas 'numpy.float64'>*)

Read in fields for first node in all trees/forest.

This function is optimized for the struct of arrays layout. It will work for array of structs layout, but field access will be 1 to 2 orders of magnitude slower.

__init__(*arbor*, *default_dtype=<class 'numpy.float64'>*)
Initialize self. See help(type(self)) for accurate signature.

Methods

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

ytree.frontends.lhalotree.io.LHaloTreeTreeFieldIO

class ytree.frontends.lhalotree.io.**LHaloTreeTreeFieldIO**(*arbor*, *de-fault_dtype=<class 'numpy.float64'>*)

__init__(*arbor*, *default_dtype=<class 'numpy.float64'>*)
Initialize self. See help(type(self)) for accurate signature.

Methods

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

ytree.frontends.lhalotree.io.LHaloTreeRootFieldIO

class ytree.frontends.lhalotree.io.**LHaloTreeRootFieldIO**(*arbor*, *de-fault_dtype=<class 'numpy.float64'>*)

__init__(*arbor*, *default_dtype=<class 'numpy.float64'>*)
Initialize self. See help(type(self)) for accurate signature.

Methods

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

ytree.frontends.lhalotree_hdf5.io.LHaloTreeHDF5TreeFieldIO

class ytree.frontends.lhalotree_hdf5.io.**LHaloTreeHDF5TreeFieldIO**(*arbor*, *default_dtype=<class 'numpy.float64'>*)

__init__(*arbor*, *default_dtype=<class 'numpy.float64'>*)
Initialize self. See help(type(self)) for accurate signature.

Methods

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

ytree.frontends.moria.io.MoriaTreeFieldIO

class ytree.frontends.moria.io.**MoriaTreeFieldIO**(*arbor*, *default_dtype=<class 'numpy.float64'>*)

__init__(*arbor*, *default_dtype=<class 'numpy.float64'>*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(arbor[, default_dtype])	Initialize self.
get_fields(data_object[, fields])	Call _setup_tree if asking for desc_uid so we can correct it.

ytree.frontends.ytree.io.YTreeTreeFieldIO

class ytree.frontends.ytree.io.**YTreeTreeFieldIO**(*arbor*, *default_dtype=<class 'numpy.float64'>*)

__init__(*arbor*, *default_dtype=<class 'numpy.float64'>*)
Initialize self. See help(type(self)) for accurate signature.

Methods

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

ytree.frontends.ytree.io.YTreeRootFieldIO

class ytree.frontends.ytree.io.**YTreeRootFieldIO**(*arbor*, *default_dtype=<class 'numpy.float64'>*)

__init__(*arbor*, *default_dtype=<class 'numpy.float64'>*)
Initialize self. See help(type(self)) for accurate signature.

Methods

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

DataFile Subclasses

Subclasses for data i/o from individual files.

`AHFDataFile`(filename, arbor)
`ConsistentTreesDataFile`(filename)
`ConsistentTreesHlistDataFile`(filename, arbor)
`ConsistentTreesHDF5DataFile`(filename, linkname)
`LHaloTreeHDF5DataFile`(filename, linkname)
`MoriaDataFile`(filename)
`RockstarDataFile`(filename, arbor)
`TreeFarmDataFile`(filename, arbor)
`YTreeDataFile`(filename)

ytree.frontends.ahf.io.AHFDataFile

class ytree.frontends.ahf.io.**AHFDataFile**(*filename*, *arbor*)

__init__(*filename*, *arbor*)
Initialize self. See help(type(self)) for accurate signature.

Methods

`__init__`(filename, arbor)	Initialize self.
`close`()	
`open`()	

Attributes

`links`

ytree.frontends.consistent_trees.io.ConsistentTreesDataFile

class ytree.frontends.consistent_trees.io.**ConsistentTreesDataFile**(*filename*)

__init__(*filename*)
Initialize self. See help(type(self)) for accurate signature.

Methods

`__init__`(filename)	Initialize self.
`close`()	
`open`()	

ytree.frontends.consistent_trees.io.ConsistentTreesHlistDataFile

class `ytree.frontends.consistent_trees.io.`**ConsistentTreesHlistDataFile**(*filename*, *arbor*)

__init__(*filename*, *arbor*)
Initialize self. See help(type(self)) for accurate signature.

Methods

`__init__`(filename, arbor)	Initialize self.
`close`()	
`open`()	

ytree.frontends.consistent_trees_hdf5.io.ConsistentTreesHDF5DataFile

class `ytree.frontends.consistent_trees_hdf5.io.`**ConsistentTreesHDF5DataFile**(*filename*, *linkname*)

__init__(*filename*, *linkname*)
Initialize self. See help(type(self)) for accurate signature.

Methods

`__init__`(filename, linkname)	Initialize self.
`close`()	
`open`()	

ytree.frontends.lhalotree_hdf5.io.LHaloTreeHDF5DataFile

class `ytree.frontends.lhalotree_hdf5.io.`**LHaloTreeHDF5DataFile**(*filename*, *linkname*)

__init__(*filename*, *linkname*)
Initialize self. See help(type(self)) for accurate signature.

Methods

`__init__`(filename, linkname)	Initialize self.
`close`()	
`open`()	

ytree.frontends.moria.io.MoriaDataFile

class `ytree.frontends.moria.io.`**`MoriaDataFile`**(*filename*)

`__init__`(*filename*)
Initialize self. See help(type(self)) for accurate signature.

Methods

`__init__`(filename)	Initialize self.
`close`()	
`open`()	
`read_data`(field, index)	

Attributes

`fh`
`field_cache`
`full_read`

ytree.frontends.rockstar.io.RockstarDataFile

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

`__init__`(*filename*, *arbor*)
Initialize self. See help(type(self)) for accurate signature.

Methods

`__init__`(filename, arbor)	Initialize self.
`close`()	
`open`()	

ytree.frontends.treefarm.io.TreeFarmDataFile

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

`__init__`(*filename*, *arbor*)
Initialize self. See help(type(self)) for accurate signature.

Methods

`__init__`(filename, arbor)	Initialize self.
`close`()	
`open`()	

ytree.frontends.ytree.io.YTreeDataFile

class `ytree.frontends.ytree.io.`**`YTreeDataFile`**(*filename*)

`__init__`(*filename*)
Initialize self. See help(type(self)) for accurate signature.

Methods

`__init__`(filename)	Initialize self.
`close`()	
`open`()	

2.16.2 ChangeLog

This is a log of changes to `ytree` over its release history.

Contributors

The CREDITS file contains the most up-to-date list of everyone who has contributed to the `ytree` source code.

Version 3.1.1

Release date: *February 3, 2022*

Bugfixes

- Allow parallel_trees to work with non-root trees. (PR #123)
- Use smarter regexes to get AHF naming scheme. (PR #118)
- Add return value to comply with yt. (PR #121)

Infrastructure Updates

- Implement _apply_units method. (PR #122)
- Enable parallelism on circleci. (PR #120)
- Create pypi upload action. (PR #124)

Version 3.1

Release date: *August 30, 2021*

New Featues

- Add AnalysisPipeline (PR #113)
- Add Parallel Iterators (PR #112)

Version 3.0

Release date: *August 3, 2021*

New Featues

- Halo selection and generation with yt data objects (PR #82)
- Add frontends for consistent-trees hlist and locations.dat files (PR #48)
- Add consistent-trees HDF5 frontend (PR #53)
- Add LHaloTree_hdf5 frontend (PR #81)
- Add TreeFrog frontend (PR #103, #95, #88)
- Add Moria frontend (PR #84)
- Add get_node and get_leaf_nodes functions (PR #80)
- Add get_root_nodes function (PR #91)
- Add add_vector_field function (PR #71)
- Add plot customization (PR #49)

Enhancements

- All functions returning TreeNodes now return generators for a significant speed and memory usage improvement. (PR #104, #64, #61)
- Speed and usability improvements to select_halos function (PR #83, #72)
- Add parallel analysis docs (PR #106)
- Make field_data an public facing attribute. (PR #105)
- Improved sorting for node_io_loop in ctrees_group and ctrees_hdf5 (PR #87)
- Relax requirements on cosmological parameters and add load options for AHF frontend (PR #76)
- Speed and usability updates to save_arbor function (PR #68, #58)
- Various infrastructure updates for newer versions of Python and dependencies (PR #92, #78, #75, #60, #54, #45)
- Update frontend development docs (PR #69)
- CI updates (PR #101, #96, #94, #93, #86, #79, #74, #73) #63, #55, #51, #50, #43, #42)
- Remove support for ytree-1.x outputs (PR #62)

- Drop support for python 3.5 (PR #59)
- Drop support for Python 2 (PR #41)

Bugfixes

- Use file sizes of loaded arbor when only saving analysis fields. (PR #100)
- Use regex for more robust filename check. (PR #77, #47)
- Fix issue with saving full arbor (PR #70)
- Check if attr is bytes or string. (PR #57)
- Fix arg in error message. (PR #56)
- Account for empty ctrees files in data files list (PR #52)

Version 2.3

Release date: *December 17, 2019*

This release marks the acceptance of the ytree paper in JOSS.

This is the last release to support Python 2.

New Features

- Add TreePlot for plotting and examples docs (PR #39)

Enhancements

- Add time field (PR #25)
- Move treefarm module to separate package (PR #28)

Version 2.2.1

Release date: *October 24, 2018*

Enhancements

- Refactor of CatalogDataFile class (PR #21)
- Simplify requirements file for docs build on readthedocs.io (PR #22)

Bugfixes

- Restore access to analysis fields for tree roots (PR #23)
- fix field access on non-root nodes when tree is not setup (PR #20)
- fix issue of uid and desc_uid fields being clobbered during initial field access (PR #19)

Version 2.2

Release date: *August 28, 2018*

New Features

- add vector fields.
- add select_halos function.

Enhancements

- significant refactor of field and i/o systems.
- upgrades to testing infrastructure.

Version 2.1.1

Release date: *April 23, 2018*

Bugfixes

- update environment.yml to fix broken readthedocs build.

Version 2.1

Release date: *April 20, 2018*

New Features

- add support for LHaloTree format.
- add support for Amiga Halo Finder format.

Version 2.0.2

Release date: *February 16, 2018*

Enhancements

- significantly improved i/o for ytree frontend.

Version 2.0

Release date: *August 07, 2017*

This is significant overhaul of the ytree machinery.

New Features

- tree building and field i/o now occur on-demand.
- support for yt-like derived fields that can be defined with simple functions.
- support for yt-like alias fields allowing for universal field naming conventions to simplify writing scripts for multiple data formats.
- support for analysis fields which allow users to save the results of expensive halo analysis to fields associated with each halo.
- all fields in consistent-trees and Rockstar now fully supported with units.
- an optimized format for saving and reloading trees for fast field access.

Enhancements

- significantly improved documentation including a guide to adding support for new file formats.

Version 1.1

Release date: *January 12, 2017*

New Features

- New, more yt-like field querying syntax for both arbors and tree nodes.

Enhancements

- Python3 now supported.
- More robust unit system with restoring of unit registries from stored json.
- Added minimum radius to halo sphere selector.
- Replaced import of yt for specific imports of all required functions.
- Added ytree logger.
- Docs updated and API reference docs added.

Bugfixes

- Allow non-root trees to be saved and reloaded.
- Fix bug allowing trees that end before the final output.

Version 1.0

Release date: *Sep 26, 2016*

The inaugural release of ytree!

CHAPTER 3

Citing ytree

If you use ytree in your work, please cite the following:

Smith et al., (2019). ytree: A Python package for analyzing merger trees. Journal of Open Source Software, 4(44), 1881, https://doi.org/10.21105/joss.01881

For BibTeX users:

```
@article{ytree,
  doi = {10.21105/joss.01881},
  url = {https://doi.org/10.21105/joss.01881},
  year  = {2019},
  month = {dec},
  publisher = {The Open Journal},
  volume = {4},
  number = {44},
  pages = {1881},
  author = {Britton D. Smith and Meagan Lang},
  title = {ytree: A Python package for analyzing merger trees},
  journal = {Journal of Open Source Software}
}
```

If you would like to also cite the specific version of `ytree` used in your work, include the following reference:

```
@software{britton_smith_2021_5336665,
  author       = {Britton Smith and
                  Meagan Lang and
                  Juanjo Bazán},
  title        = {ytree-project/ytree: ytree 3.1 Release},
  month        = aug,
  year         = 2021,
  publisher    = {Zenodo},
  version      = {ytree-3.1.0},
  doi          = {10.5281/zenodo.5336665},
  url          = {https://doi.org/10.5281/zenodo.5336665}
}
```

CHAPTER 4

Search

- search

Index

Symbols

`__init__()` (*ytree.analysis.analysis_operators.AnalysisOperation method*), 58
`__init__()` (*ytree.analysis.analysis_pipeline.AnalysisPipeline method*), 57
`__init__()` (*ytree.data_structures.arbor.Arbor method*), 42
`__init__()` (*ytree.data_structures.arbor.CatalogArbor method*), 63
`__init__()` (*ytree.data_structures.arbor.SegmentedArbor method*), 62
`__init__()` (*ytree.data_structures.detection.Detector method*), 64
`__init__()` (*ytree.data_structures.detection.FieldDetector method*), 65
`__init__()` (*ytree.data_structures.detection.SelectionDetector method*), 66
`__init__()` (*ytree.data_structures.fields.FieldContainer method*), 68
`__init__()` (*ytree.data_structures.fields.FieldInfoContainer method*), 67
`__init__()` (*ytree.data_structures.io.CatalogDataFile method*), 70
`__init__()` (*ytree.data_structures.io.DataFile method*), 69
`__init__()` (*ytree.data_structures.io.DefaultRootFieldIO method*), 69
`__init__()` (*ytree.data_structures.io.FieldIO method*), 68
`__init__()` (*ytree.data_structures.io.TreeFieldIO method*), 69
`__init__()` (*ytree.data_structures.tree_node.TreeNode method*), 47
`__init__()` (*ytree.data_structures.tree_node_selector.TreeNodeSelector method*), 50
`__init__()` (*ytree.frontends.ahf.arbor.AHFArbor method*), 70
`__init__()` (*ytree.frontends.ahf.fields.AHFFieldInfo method*), 82
`__init__()` (*ytree.frontends.ahf.io.AHFDataFile method*), 94
`__init__()` (*ytree.frontends.consistent_trees.arbor.ConsistentTreesArbo method*), 71
`__init__()` (*ytree.frontends.consistent_trees.arbor.ConsistentTreesGrou method*), 72
`__init__()` (*ytree.frontends.consistent_trees.arbor.ConsistentTreesHlist method*), 73
`__init__()` (*ytree.frontends.consistent_trees.fields.ConsistentTreesField method*), 83
`__init__()` (*ytree.frontends.consistent_trees.io.ConsistentTreesDataFile method*), 94
`__init__()` (*ytree.frontends.consistent_trees.io.ConsistentTreesHlistDat method*), 95
`__init__()` (*ytree.frontends.consistent_trees.io.ConsistentTreesTreeFiel method*), 91
`__init__()` (*ytree.frontends.consistent_trees_hdf5.arbor.ConsistentTrees method*), 74
`__init__()` (*ytree.frontends.consistent_trees_hdf5.fields.ConsistentTrees method*), 84
`__init__()` (*ytree.frontends.consistent_trees_hdf5.io.ConsistentTreesHD method*), 95
`__init__()` (*ytree.frontends.consistent_trees_hdf5.io.ConsistentTreesHD method*), 92
`__init__()` (*ytree.frontends.consistent_trees_hdf5.io.ConsistentTreesHD method*), 91
`__init__()` (*ytree.frontends.lhalotree.arbor.LHaloTreeArbor method*), 75
`__init__()` (*ytree.frontends.lhalotree.fields.LHaloTreeFieldInfo method*), 85
`__init__()` (*ytree.frontends.lhalotree.io.LHaloTreeRootFieldIO method*), 92
`__init__()` (*ytree.frontends.lhalotree.io.LHaloTreeTreeFieldIO method*), 92
`__init__()` (*ytree.frontends.lhalotree_hdf5.arbor.LHaloTreeHDF5Arbo method*), 76
`__init__()` (*ytree.frontends.lhalotree_hdf5.fields.LHaloTreeHDF5Field method*), 86
`__init__()` (*ytree.frontends.lhalotree_hdf5.io.LHaloTreeHDF5DataFile method*), 95

`__init__()` (*ytree.frontends.lhalotree_hdf5.io.LHaloTreeHDF5TreeFieldIO method*), 93
`__init__()` (*ytree.frontends.moria.arbor.MoriaArbor method*), 77
`__init__()` (*ytree.frontends.moria.fields.MoriaFieldInfo method*), 87
`__init__()` (*ytree.frontends.moria.io.MoriaDataFile method*), 96
`__init__()` (*ytree.frontends.moria.io.MoriaTreeFieldIO method*), 93
`__init__()` (*ytree.frontends.rockstar.arbor.RockstarArbor method*), 78
`__init__()` (*ytree.frontends.rockstar.fields.RockstarFieldInfo method*), 88
`__init__()` (*ytree.frontends.rockstar.io.RockstarDataFile method*), 96
`__init__()` (*ytree.frontends.treefarm.arbor.TreeFarmArbor method*), 79
`__init__()` (*ytree.frontends.treefarm.fields.TreeFarmFieldInfo method*), 89
`__init__()` (*ytree.frontends.treefarm.io.TreeFarmDataFile method*), 96
`__init__()` (*ytree.frontends.ytree.arbor.YTreeArbor method*), 80
`__init__()` (*ytree.frontends.ytree.io.YTreeDataFile method*), 97
`__init__()` (*ytree.frontends.ytree.io.YTreeRootFieldIO method*), 93
`__init__()` (*ytree.frontends.ytree.io.YTreeTreeFieldIO method*), 93
`__init__()` (*ytree.visualization.tree_plot.TreePlot method*), 55

A

`add_alias_field()` (*ytree.data_structures.arbor.Arbor method*), 43
`add_analysis_field()` (*ytree.data_structures.arbor.Arbor method*), 43
`add_derived_field()` (*ytree.data_structures.arbor.Arbor method*), 44
`add_operation()` (*ytree.analysis.analysis_pipeline.AnalysisPipeline method*), 57
`add_recipe()` (*ytree.analysis.analysis_pipeline.AnalysisPipeline method*), 58
`add_tree_node_selector()` (*in module ytree.data_structures.tree_node_selector*), 50
`add_vector_field()` (*ytree.data_structures.arbor.Arbor method*), 44
`AHFArbor` (*class in ytree.frontends.ahf.arbor*), 70
`AHFDataFile` (*class in ytree.frontends.ahf.io*), 94
`AHFFieldInfo` (*class in ytree.frontends.ahf.fields*), 82
`AnalysisOperation` (*class in ytree.analysis.analysis_operators*), 58
`AnalysisPipeline` (*class in ytree.analysis.analysis_pipeline*), 56
`Arbor` (*class in ytree.data_structures.arbor*), 42

C

`CatalogArbor` (*class in ytree.data_structures.arbor*), 63
`CatalogDataFile` (*class in ytree.data_structures.io*), 70
`ConsistentTreesArbor` (*class in ytree.frontends.consistent_trees.arbor*), 71
`ConsistentTreesDataFile` (*class in ytree.frontends.consistent_trees.io*), 94
`ConsistentTreesFieldInfo` (*class in ytree.frontends.consistent_trees.fields*), 83
`ConsistentTreesGroupArbor` (*class in ytree.frontends.consistent_trees.arbor*), 72
`ConsistentTreesHDF5Arbor` (*class in ytree.frontends.consistent_trees_hdf5.arbor*), 74
`ConsistentTreesHDF5DataFile` (*class in ytree.frontends.consistent_trees_hdf5.io*), 95
`ConsistentTreesHDF5FieldInfo` (*class in ytree.frontends.consistent_trees_hdf5.fields*), 84
`ConsistentTreesHDF5RootFieldIO` (*class in ytree.frontends.consistent_trees_hdf5.io*), 92
`ConsistentTreesHDF5TreeFieldIO` (*class in ytree.frontends.consistent_trees_hdf5.io*), 91
`ConsistentTreesHlistArbor` (*class in ytree.frontends.consistent_trees.arbor*), 73
`ConsistentTreesHlistDataFile` (*class in ytree.frontends.consistent_trees.io*), 95
`ConsistentTreesTreeFieldIO` (*class in ytree.frontends.consistent_trees.io*), 91

D

`DataFile` (*class in ytree.data_structures.io*), 69
`DefaultRootFieldIO` (*class in ytree.data_structures.io*), 69
`Detector` (*class in ytree.data_structures.detection*), 64

F

`FieldContainer` (*class in ytree.data_structures.fields*), 68
`FieldDetector` (*class in ytree.data_structures.detection*), 65
`FieldInfoContainer` (*class in ytree.data_structures.fields*), 67
`FieldIO` (*class in ytree.data_structures.io*), 68

G

`get_leaf_nodes()` (*ytree.data_structures.tree_node.TreeNode method*), 47

get_node() (*ytree.data_structures.tree_node.TreeNode method*), 48
get_nodes_from_selection() (*ytree.frontends.ytree.arbor.YTreeArbor method*), 53
get_root_nodes() (*ytree.data_structures.tree_node.TreeNode method*), 48
get_yt_selection() (*ytree.frontends.ytree.arbor.YTreeArbor method*), 51

L

LHaloTreeArbor (*class in ytree.frontends.lhalotree.arbor*), 75
LHaloTreeFieldInfo (*class in ytree.frontends.lhalotree.fields*), 85
LHaloTreeHDF5Arbor (*class in ytree.frontends.lhalotree_hdf5.arbor*), 76
LHaloTreeHDF5DataFile (*class in ytree.frontends.lhalotree_hdf5.io*), 95
LHaloTreeHDF5FieldInfo (*class in ytree.frontends.lhalotree_hdf5.fields*), 86
LHaloTreeHDF5TreeFieldIO (*class in ytree.frontends.lhalotree_hdf5.io*), 93
LHaloTreeRootFieldIO (*class in ytree.frontends.lhalotree.io*), 92
LHaloTreeTreeFieldIO (*class in ytree.frontends.lhalotree.io*), 92
load() (*in module ytree.data_structures.load*), 41

M

max_field_value() (*in module ytree.data_structures.tree_node_selector*), 50
min_field_value() (*in module ytree.data_structures.tree_node_selector*), 51
min_mass (*ytree.visualization.tree_plot.TreePlot attribute*), 54
min_mass_ratio (*ytree.visualization.tree_plot.TreePlot attribute*), 55
MoriaArbor (*class in ytree.frontends.moria.arbor*), 77
MoriaDataFile (*class in ytree.frontends.moria.io*), 96
MoriaFieldInfo (*class in ytree.frontends.moria.fields*), 87
MoriaTreeFieldIO (*class in ytree.frontends.moria.io*), 93

P

parallel_nodes() (*in module ytree.utilities.parallel*), 61
parallel_tree_nodes() (*in module ytree.utilities.parallel*), 60
parallel_trees() (*in module ytree.utilities.parallel*), 59
process_target() (*ytree.analysis.analysis_pipeline.AnalysisPipeline method*), 58

R

RockstarArbor (*class in ytree.frontends.rockstar.arbor*), 78
RockstarDataFile (*class in ytree.frontends.rockstar.io*), 96
RockstarFieldInfo (*class in ytree.frontends.rockstar.fields*), 88

S

save() (*ytree.visualization.tree_plot.TreePlot method*), 56
save_arbor() (*ytree.data_structures.arbor.Arbor method*), 45
save_tree() (*ytree.data_structures.tree_node.TreeNode method*), 49
SegmentedArbor (*class in ytree.data_structures.arbor*), 62
select_halos() (*ytree.data_structures.arbor.Arbor method*), 46
SelectionDetector (*class in ytree.data_structures.detection*), 66
set_selector() (*ytree.data_structures.arbor.Arbor method*), 46
size_field (*ytree.visualization.tree_plot.TreePlot attribute*), 54
size_log (*ytree.visualization.tree_plot.TreePlot attribute*), 54

T

TreeFarmArbor (*class in ytree.frontends.treefarm.arbor*), 79
TreeFarmDataFile (*class in ytree.frontends.treefarm.io*), 96
TreeFarmFieldInfo (*class in ytree.frontends.treefarm.fields*), 89
TreeFieldIO (*class in ytree.data_structures.io*), 69
TreeNode (*class in ytree.data_structures.tree_node*), 47
TreeNodeSelector (*class in ytree.data_structures.tree_node_selector*), 49
TreePlot (*class in ytree.visualization.tree_plot*), 54

Y

ytds (*ytree.frontends.ytree.arbor.YTreeArbor attribute*), 53
YTreeArbor (*class in ytree.frontends.ytree.arbor*), 80
YTreeDataFile (*class in ytree.frontends.ytree.io*), 97
YTreeRootFieldIO (*class in ytree.frontends.ytree.io*), 93

YTreeTreeFieldIO (*class in ytree.frontends.ytree.io*), 93