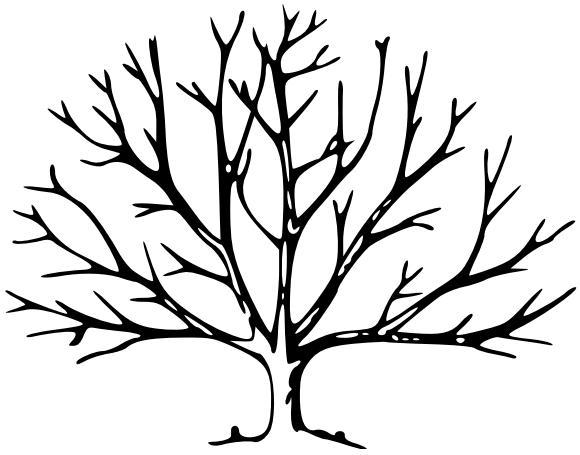


asdf Documentation

Release 2.12.1.dev90+g784740b

The ASDF Developers <help@stsci.edu>

Aug 12, 2022

CONTENTS

`asdf` is a tool for reading and writing Advanced Scientific Data Format (ASDF) files.

The **A**dvanced **S**cientific **D**ata **F**ormat (ASDF) is a next-generation interchange format for scientific data. This package contains the Python implementation of the ASDF Standard. More information on the ASDF Standard itself can be found here.

The ASDF format has the following features:

- A hierarchical, human-readable metadata format (implemented using YAML)
- Numerical arrays are stored as binary data blocks which can be memory mapped. Data blocks can optionally be compressed.
- The structure of the data can be automatically validated using schemas (implemented using JSON Schema)
- Native Python data types (numerical types, strings, dicts, lists) are serialized automatically
- ASDF can be extended to serialize custom data types

Note: This is the **A**dvanced **S**cientific **D**ata **F**ormat - if you are looking for the **A**daptable **S**eismic **D**ata **F**ormat, go here: http://seismic-data.org/

Part I

Getting Started

CHAPTER

ONE

INSTALLATION

There are several different ways to install the asdf package. Each is described in detail below.

1.1 Requirements

The asdf package has several dependencies which are listed in the project's build configuration (`setup.cfg` / `pyproject.toml`). All dependencies are available on pypi and will be automatically installed along with asdf.

Support for units, time, and transform tags requires an implementation of these types. One recommended option is the astropy package.

Optional support for lz4 compression is provided by the lz4 package.

1.2 Installing with pip

Stable releases of the ASDF Python package are registered at PyPi. The latest stable version can be installed using `pip`:

```
$ pip install asdf
```

1.3 Installing with conda

asdf is also distributed as a conda package via the conda-forge channel. It is also available through the astroconda channel.

To install asdf within an existing conda environment:

```
$ conda install -c conda-forge asdf
```

To create a new conda environment and install asdf:

```
$ conda create -n new-env-name -c conda-forge python asdf
```

1.4 Building from source

The latest development version of ASDF is available from the `master` branch on github. To clone the project:

```
$ git clone https://github.com/asdf-format/asdf
```

To install:

```
$ cd asdf
$ pip install .
```

To install in development mode:

```
$ pip install -e .
```

1.5 Running the tests

To install the test dependencies from a source checkout of the repository:

```
$ pip install -e ".[tests]"
```

To run the unit tests from a source checkout of the repository:

```
$ pytest
```

It is also possible to run the test suite from an installed version of the package.

```
$ pip install "asdf[tests]"
$ pytest --pyargs asdf
```

It is also possible to run the tests using tox.

```
$ pip install tox
```

To list all available environments:

```
$ tox -va
```

To run a specific environment:

```
$ tox -e <envname>
```

CHAPTER

TWO

OVERVIEW

Let's start by taking a look at a few basic ASDF use cases. This will introduce you to some of the core features of ASDF and will show you how to get started with using ASDF in your own projects.

To follow along with this tutorial, you will need to install the `asdf` package. See *Installation* for details.

2.1 Hello World

At its core, ASDF is a way of saving nested data structures to YAML. Here we save a `dict` with the key/value pair `'hello': 'world'`.

```
from asdf import AsdfFile

# Make the tree structure, and create a AsdfFile from it.
tree = {'hello': 'world'}
ff = AsdfFile(tree)
ff.write_to("test.asdf")

# You can also make the AsdfFile first, and modify its tree directly:
ff = AsdfFile()
ff.tree['hello'] = 'world'
ff.write_to("test.asdf")
```

test.asdf

```
#ASDF 1.0.0
#ASDF_STANDARD 1.5.0
%YAML 1.1
%TAG ! tag:stsci.edu:asdf/
--- !core/asdf-1.1.0
asdf_library: !core/software-1.0.0 {author: The ASDF Developers, homepage: 'http://github.
↪com/asdf-format/asdf',
  name: asdf, version: 2.12.1.dev90+g784740b}
history:
  extensions:
  - !core/extension_metadata-1.0.0
    extension_class: asdf.extension.BuiltinExtension
    software: !core/software-1.0.0 {name: asdf, version: 2.12.1.dev90+g784740b}
hello: world
...
```

2.2 Creating Files

We're going to store several `numpy` arrays and other data to an ASDF file. We do this by creating a "tree", which is simply a `dict`, and we provide it as input to the constructor of `AsdfFile`:

```
import asdf
import numpy as np

# Create some data
sequence = np.arange(100)
squares  = sequence**2
random = np.random.random(100)

# Store the data in an arbitrarily nested dictionary
tree = {
    'foo': 42,
    'name': 'Monty',
    'sequence': sequence,
    'powers': { 'squares' : squares },
    'random': random
}

# Create the ASDF file object from our data tree
af = asdf.AsdfFile(tree)

# Write the data to a new file
af.write_to('example.asdf')
```

If we open the newly created file's metadata section, we can see some of the key features of ASDF on display:

example.asdf

```
#ASDF 1.0.0
#ASDF_STANDARD 1.5.0
%YAML 1.1
%TAG ! tag:stsci.edu:asdf/
--- !core/asdf-1.1.0
asdf_library: !core/software-1.0.0 {author: The ASDF Developers, homepage: 'http://github.
↪com/asdf-format/asdf',
  name: asdf, version: 2.12.1.dev90+g784740b}
history:
  extensions:
  - !core/extension_metadata-1.0.0
    extension_class: asdf.extension.BuiltinExtension
    software: !core/software-1.0.0 {name: asdf, version: 2.12.1.dev90+g784740b}
foo: 42
name: Monty
powers:
  squares: !core/ndarray-1.0.0
```

(continues on next page)

(continued from previous page)

```
    source: 1
    datatype: int64
    byteorder: little
    shape: [100]
random: !core/ndarray-1.0.0
  source: 2
  datatype: float64
  byteorder: little
  shape: [100]
sequence: !core/ndarray-1.0.0
  source: 0
  datatype: int64
  byteorder: little
  shape: [100]
...
```

The metadata in the file mirrors the structure of the tree that was stored. It is hierarchical and human-readable. Notice that metadata has been added to the tree that was not explicitly given by the user. Notice also that the numerical array data is not stored in the metadata tree itself. Instead, it is stored as binary data blocks below the metadata section (not shown above).

A rendering of the binary data contained in the file can be found below. Observe that the value of `source` in the metadata corresponds to the block number (e.g. `BLOCK 0`) of the block which contains the binary data.

example.asdf

```
BLOCK 0:
    allocated_size: 800
    used_size: 800
    data_size: 800
    data: b'0000000000000000010000000000000002000000...'
```

```
BLOCK 1:
    allocated_size: 800
    used_size: 800
    data_size: 800
    data: b'0000000000000000010000000000000004000000...'
```

```
BLOCK 2:
    allocated_size: 800
    used_size: 800
    data_size: 800
    data: b'888f1c5ab8c5d33fe490ceb84845c43fa092e573...'
```

```
#ASDF BLOCK INDEX
%YAML 1.1
---
- 767
- 1621
- 2475
...
```

It is possible to compress the array data when writing the file:

```
af.write_to('compressed.asdf', all_array_compression='zlib')
```

The built-in compression algorithms are `'zlib'`, and `'bzp2'`. The `'lz4'` algorithm becomes available when the lz4 package is installed. Other compression algorithms may be available via extensions.

2.3 Reading Files

To read an existing ASDF file, we simply use the top-level `open` function of the `asdf` package:

```
import asdf

af = asdf.open('example.asdf')
```

The `open` function also works as a context handler:

```
with asdf.open('example.asdf') as af:
    ...
```

To get a quick overview of the data stored in the file, use the top-level `AsdfFile.info()` method:

```
>>> import asdf
>>> af = asdf.open('example.asdf')
>>> af.info()
root (AsdfObject)
├─asdf_library (Software)
│ ├─author (str): The ASDF Developers
│ ├─homepage (str): http://github.com/asdf-format/asdf
│ ├─name (str): asdf
│ └─version (str): 2.8.0
├─history (dict)
│ └─extensions (list)
│   └─[0] (ExtensionMetadata)
│     ├─extension_class (str): asdf.extension.BuiltinExtension
│     └─software (Software)
│       ├─name (str): asdf
│       └─version (str): 2.8.0
├─foo (int): 42
├─name (str): Monty
├─powers (dict)
│ └─squares (NDArrayType): shape=(100,), dtype=int64
├─random (NDArrayType): shape=(100,), dtype=float64
└─sequence (NDArrayType): shape=(100,), dtype=int64
```

The `AsdfFile` behaves like a Python `dict`, and nodes are accessed like any other dictionary entry:

```
>>> af['name']
'Monty'
>>> af['powers']
{'squares': <array (unloaded) shape: [100] dtype: int64>}
```

Array data remains unloaded until it is explicitly accessed:

```
>>> af['powers']['squares']
array([   0,    1,    4,    9,   16,   25,   36,   49,   64,   81,  100,
        121,  144,  169,  196,  225,  256,  289,  324,  361,  400,  441,
        484,  529,  576,  625,  676,  729,  784,  841,  900,  961, 1024,
       1089, 1156, 1225, 1296, 1369, 1444, 1521, 1600, 1681, 1764, 1849,
       1936, 2025, 2116, 2209, 2304, 2401, 2500, 2601, 2704, 2809, 2916,
       3025, 3136, 3249, 3364, 3481, 3600, 3721, 3844, 3969, 4096, 4225,
       4356, 4489, 4624, 4761, 4900, 5041, 5184, 5329, 5476, 5625, 5776,
       5929, 6084, 6241, 6400, 6561, 6724, 6889, 7056, 7225, 7396, 7569,
       7744, 7921, 8100, 8281, 8464, 8649, 8836, 9025, 9216, 9409, 9604,
       9801])

>>> import numpy as np
>>> expected = [x**2 for x in range(100)]
>>> np.equal(af['powers']['squares'], expected).all()
True
```

By default, uncompressed data blocks are memory mapped for efficient access. Memory mapping can be disabled by using the `copy_arrays` option of `open` when reading:

```
af = asdf.open('example.asdf', copy_arrays=True)
```

CHAPTER

THREE

CORE FEATURES

This section discusses the core features of the ASDF data format, and provides examples and use cases that are specific to the Python implementation.

3.1 Data Model

The fundamental data object in ASDF is the `tree`, which is a nested combination of basic data structures: dictionaries, lists, strings and numbers. In Python, these types correspond to `dict`, `list`, `str`, and `int`, `float`, and `complex`, respectively. The top-level tree object behaves like a Python dictionary and supports arbitrary nesting of data structures. For simple examples of creating and reading trees, see *Overview*.

Note: The ASDF Standard imposes a maximum size of 64-bit signed integers literals in the tree (see Literal integer values in the Tree for details and justification). Attempting to store a larger value as a YAML literal will result in a validation error.

For arbitrary precision integer support, see `IntegerType`.

Integers and floats of up to 64 bits can be stored inside of `numpy` arrays (see below).

One of the key features of `asdf` is its ability to serialize `numpy` arrays. This is discussed in detail in *Array Data*.

While the core `asdf` package supports serialization of basic data types and Numpy arrays, its true power comes from its ability to be extended to support serialization of a wide range of custom data types. Details on using ASDF extensions can be found in *Using extensions*. Details on creating custom ASDF extensions to support custom data types can be found in *Extending ASDF*.

3.2 Array Data

Much of ASDF's power and convenience comes from its ability to represent multidimensional array data. The `asdf` Python package provides native support for `numpy` arrays.

3.2.1 Saving arrays

Beyond the basic data types of dictionaries, lists, strings and numbers, the most important thing ASDF can save is arrays. It's as simple as putting a `numpy` array somewhere in the tree. Here, we save an 8x8 array of random floating-point numbers (using `numpy.random.rand`). Note that the resulting YAML output contains information about the structure (size and data type) of the array, but the actual array content is in a binary block.

```
from asdf import AsdfFile
import numpy as np

tree = {'my_array': np.random.rand(8, 8)}
ff = AsdfFile(tree)
ff.write_to("test.asdf")
```

Note: In the file examples below, the first YAML part appears as it appears in the file. The `BLOCK` sections are stored as binary data in the file, but are presented in human-readable form on this page.

test.asdf

```
#ASDF 1.0.0
#ASDF_STANDARD 1.5.0
%YAML 1.1
%TAG ! tag:stsci.edu:asdf/
--- !core/asdf-1.1.0
asdf_library: !core/software-1.0.0 {author: The ASDF Developers, homepage: 'http://github.
↪com/asdf-format/asdf',
  name: asdf, version: 2.12.1.dev90+g784740b}
history:
  extensions:
  - !core/extension_metadata-1.0.0
    extension_class: asdf.extension.BuiltinExtension
    software: !core/software-1.0.0 {name: asdf, version: 2.12.1.dev90+g784740b}
my_array: !core/ndarray-1.0.0
  source: 0
  datatype: float64
  byteorder: little
  shape: [8, 8]
...
```

```
BLOCK 0:
    allocated_size: 512
    used_size: 512
    data_size: 512
    data: b'bcf5561db8ebea3fa442326a6aebdb3f88ca5779...'
```

```
#ASDF BLOCK INDEX
%YAML 1.1
---
- 543
...
```

3.2.2 Sharing of data

Arrays that are views on the same data automatically share the same data in the file. In this example an array and a subview on that same array are saved to the same file, resulting in only a single block of data being saved.

```
from asdf import AsdfFile
import numpy as np

my_array = np.random.rand(8, 8)
subset = my_array[2:4,3:6]
tree = {
    'my_array': my_array,
    'subset':   subset
}
ff = AsdfFile(tree)
ff.write_to("test.asdf")
```

test.asdf

```
#ASDF 1.0.0
#ASDF_STANDARD 1.5.0
%YAML 1.1
%TAG ! tag:stsci.edu:asdf/
--- !core/asdf-1.1.0
asdf_library: !core/software-1.0.0 {author: The ASDF Developers, homepage: 'http://github.
↪com/asdf-format/asdf',
  name: asdf, version: 2.12.1.dev90+g784740b}
history:
  extensions:
  - !core/extension_metadata-1.0.0
    extension_class: asdf.extension.BuiltinExtension
    software: !core/software-1.0.0 {name: asdf, version: 2.12.1.dev90+g784740b}
my_array: !core/ndarray-1.0.0
  source: 0
  datatype: float64
  byteorder: little
  shape: [8, 8]
subset: !core/ndarray-1.0.0
  source: 0
  datatype: float64
  byteorder: little
  shape: [2, 3]
  offset: 152
  strides: [64, 8]
...
```

```
BLOCK 0:
    allocated_size: 512
    used_size: 512
    data_size: 512
    data: b'e8f1b7e618b3e33ffa34994bb531e03f00c508b7...'
```

```
#ASDF BLOCK INDEX
%YAML 1.1
---
- 672
...
```

3.2.3 Saving inline arrays

For small arrays, you may not care about the efficiency of a binary representation and just want to save the array contents directly in the YAML tree. The `set_array_storage` method can be used to set the storage type of the associated data. The allowed values are `internal`, `external`, and `inline`.

- `internal`: The default. The array data will be stored in a binary block in the same ASDF file.
- `external`: Store the data in a binary block in a separate ASDF file (also known as "exploded" format, which discussed below in *Saving external arrays*).
- `inline`: Store the data as YAML inline in the tree.

```
from asdf import AsdfFile
import numpy as np

my_array = np.random.rand(8, 8)
tree = {'my_array': my_array}
ff = AsdfFile(tree)
ff.set_array_storage(my_array, 'inline')
ff.write_to("test.asdf")
```

test.asdf

```
#ASDF 1.0.0
#ASDF_STANDARD 1.5.0
%YAML 1.1
%TAG ! tag:stsci.edu:asdf/
--- !core/asdf-1.1.0
asdf_library: !core/software-1.0.0 {author: The ASDF Developers, homepage: 'http://github.
↪com/asdf-format/asdf',
  name: asdf, version: 2.12.1.dev90+g784740b}
history:
  extensions:
  - !core/extension_metadata-1.0.0
    extension_class: asdf.extension.BuiltinExtension
    software: !core/software-1.0.0 {name: asdf, version: 2.12.1.dev90+g784740b}
my_array: !core/ndarray-1.0.0
  data:
  - [0.24716250343567903, 0.22748466105500187, 0.42188087889618253, 0.3050079694743013,
    0.05866694738345257, 0.5761751260775555, 0.5314300995064118, 0.4976755709341779]
  - [0.7226563089981942, 0.01768851330486121, 0.6179849244167936, 0.5928452435638892,
    0.6001118249651083, 0.8649735978164809, 0.2728302177992947, 0.7888576667354937]
  - [0.09190825672690361, 0.4442513085465809, 0.9139179271466382, 0.6473659746202919,
    0.24020031891326388, 0.9847421527874193, 0.9082664927945632, 0.8412496220370519]
```

(continues on next page)

(continued from previous page)

```
  - [0.39609124766050363, 0.9249505634559739, 0.7225998105677932, 0.4935532781745239,
    0.007506096578081101, 0.38842049158836844, 0.339170506479325, 0.9394753477162989]
  - [0.043656987255935475, 0.34524472225114566, 0.7579965834194343, 0.8238741699444497,
    0.35925702368131596, 0.2577240317539323, 0.797436031040072, 0.5091205690967755]
  - [0.9883789491587976, 0.37026491362735225, 0.3301079801700705, 0.3351939167137603,
    0.018936079563036667, 0.8747024758808037, 0.27873909559412025, 0.3734498035955689]
  - [0.9828606521410929, 0.10366827776956722, 0.2304554628424048, 0.8014656854808543,
    0.7154535678421137, 0.14929757452639691, 0.027555746540530945, 0.7084772029109591]
  - [0.14083229703981426, 0.005678775190763208, 0.05350946396939049, 0.7931690915963353,
    0.7302284618407466, 0.05154795047363048, 0.43957232943993974, 0.7287218405862054]
  datatype: float64
  shape: [8, 8]
...
```

Alternatively, it is possible to use the `all_array_storage` parameter of `AsdfFile.write_to` and `AsdfFile.update` to control the storage format of all arrays in the file.

```
# This controls the output format of all arrays in the file
ff.write_to("test.asdf", all_array_storage='inline')
```

For automatic management of the array storage type based on number of elements, see *array_inline_threshold*.

3.2.4 Saving external arrays

ASDF files may also be saved in "exploded form", which creats multiple files corresponding to the following data items:

- One ASDF file containing only the header and tree.
- *n* ASDF files, each containing a single array data block.

Exploded form is useful in the following scenarios:

- Over a network protocol, such as HTTP, a client may only need to access some of the blocks. While reading a subset of the file can be done using HTTP `Range` headers, it still requires one (small) request per block to "jump" through the file to determine the start location of each block. This can become time-consuming over a high-latency network if there are many blocks. Exploded form allows each block to be requested directly by a specific URI.
- An ASDF writer may stream a table to disk, when the size of the table is not known at the outset. Using exploded form simplifies this, since a standalone file containing a single table can be iteratively appended to without worrying about any blocks that may follow it.

To save a block in an external file, set its block type to `'external'`.

```
from asdf import AsdfFile
import numpy as np

my_array = np.random.rand(8, 8)
tree = {'my_array': my_array}
ff = AsdfFile(tree)

# On an individual block basis:
```

(continues on next page)

(continued from previous page)

```
ff.set_array_storage(my_array, 'external')
ff.write_to("test.asdf")

# Or for every block:
ff.write_to("test.asdf", all_array_storage='external')
```

test.asdf

```
#ASDF 1.0.0
#ASDF_STANDARD 1.5.0
%YAML 1.1
%TAG ! tag:stsci.edu:asdf/
--- !core/asdf-1.1.0
asdf_library: !core/software-1.0.0 {author: The ASDF Developers, homepage: 'http://github.
↪com/asdf-format/asdf',
  name: asdf, version: 2.12.1.dev90+g784740b}
history:
  extensions:
  - !core/extension_metadata-1.0.0
    extension_class: asdf.extension.BuiltinExtension
    software: !core/software-1.0.0 {name: asdf, version: 2.12.1.dev90+g784740b}
my_array: !core/ndarray-1.0.0
  source: test0000.asdf
  datatype: float64
  byteorder: little
  shape: [8, 8]
...
```

test0000.asdf

```
#ASDF 1.0.0
#ASDF_STANDARD 1.5.0
%YAML 1.1
%TAG ! tag:stsci.edu:asdf/
--- !core/asdf-1.1.0
asdf_library: !core/software-1.0.0 {author: The ASDF Developers, homepage: 'http://github.
↪com/asdf-format/asdf',
  name: asdf, version: 2.12.1.dev90+g784740b}
history:
  extensions:
  - !core/extension_metadata-1.0.0
    extension_class: asdf.extension.BuiltinExtension
    software: !core/software-1.0.0 {name: asdf, version: 2.12.1.dev90+g784740b}
...
```

```
BLOCK 0:
    allocated_size: 512
    used_size: 512
    data_size: 512
```

(continues on next page)

(continued from previous page)

```
    data: b'4477ba5673c5cd3fac02fb92a603ce3f70507313...'
```

```
#ASDF BLOCK INDEX
%YAML 1.1
---
- 445
...
```

3.2.5 Streaming array data

In certain scenarios, you may want to stream data to disk, rather than writing an entire array of data at once. For example, it may not be possible to fit the entire array in memory, or you may want to save data from a device as it comes in to prevent data loss. The ASDF Standard allows exactly one streaming block per file where the size of the block isn't included in the block header, but instead is implicitly determined to include all of the remaining contents of the file. By definition, it must be the last block in the file.

To use streaming, rather than including a Numpy array object in the tree, you include a `asdf.Stream` object which sets up the structure of the streamed data, but will not write out the actual content. The file handle's `write` method is then used to manually write out the binary data.

```
from asdf import AsdfFile, Stream
import numpy as np

tree = {
    # Each "row" of data will have 128 entries.
    'my_stream': Stream([128], np.float64)
}

ff = AsdfFile(tree)
with open('test.asdf', 'wb') as fd:
    ff.write_to(fd)
    # Write 100 rows of data, one row at a time.  ``write``
    # expects the raw binary bytes, not an array, so we use
    # ``tobytes()``.
    for i in range(100):
        fd.write(np.array([i] * 128, np.float64).tobytes())
```

test.asdf

```
#ASDF 1.0.0
#ASDF_STANDARD 1.5.0
%YAML 1.1
%TAG ! tag:stsci.edu:asdf/
--- !core/asdf-1.1.0
asdf_library: !core/software-1.0.0 {author: The ASDF Developers, homepage: 'http://github.
↪com/asdf-format/asdf',
  name: asdf, version: 2.12.1.dev90+g784740b}
history:
  extensions:
```

(continues on next page)

(continued from previous page)

```
  - !core/extension_metadata-1.0.0
    extension_class: asdf.extension.BuiltinExtension
    software: !core/software-1.0.0 {name: asdf, version: 2.12.1.dev90+g784740b}
my_stream: !core/ndarray-1.0.0
  source: -1
  datatype: float64
  byteorder: little
  shape: ['*', 128]
...
```

```
BLOCK 0:
    flags: BLOCK_FLAG_STREAMED
    allocated_size: 0
    used_size: 0
    data_size: 0
    data: b'00000000000000000000000000000000000000000...'
```

A case where streaming may be useful is when converting large data sets from a different format into ASDF. In these cases it would be impractical to hold all of the data in memory as an intermediate step. Consider the following example that streams a large CSV file containing rows of integer data and converts it to numpy arrays stored in ASDF:

```
import csv
import numpy as np
from asdf import AsdfFile, Stream

tree = {
    # We happen to know in advance that each row in the CSV has 100 ints
    'data': Stream([100], np.int64)
}

ff = AsdfFile(tree)
# open the output file handle
with open('new_file.asdf', 'wb') as fd:
    ff.write_to(fd)
    # open the CSV file to be converted
    with open('large_file.csv', 'r') as cfd:
        # read each line of the CSV file
        reader = csv.reader(cfd)
        for row in reader:
            # convert each row to a numpy array
            array = np.array([int(x) for x in row], np.int64)
            # write the array to the output file handle
            fd.write(array.tobytes())
```

3.2.6 Compression

Individual blocks in an ASDF file may be compressed.

You can easily zlib or bzip2 compress all blocks:

```
from asdf import AsdfFile
import numpy as np

tree = {
    'a': np.random.rand(32, 32),
    'b': np.random.rand(64, 64)
}

target = AsdfFile(tree)
target.write_to('target.asdf', all_array_compression='zlib')
target.write_to('target.asdf', all_array_compression='bzp2')
```

target.asdf

```
#ASDF 1.0.0
#ASDF_STANDARD 1.5.0
%YAML 1.1
%TAG ! tag:stsci.edu:asdf/
--- !core/asdf-1.1.0
asdf_library: !core/software-1.0.0 {author: The ASDF Developers, homepage: 'http://github.
↪com/asdf-format/asdf',
  name: asdf, version: 2.12.1.dev90+g784740b}
history:
  extensions:
  - !core/extension_metadata-1.0.0
    extension_class: asdf.extension.BuiltinExtension
    software: !core/software-1.0.0 {name: asdf, version: 2.12.1.dev90+g784740b}
a: !core/ndarray-1.0.0
  source: 0
  datatype: float64
  byteorder: little
  shape: [32, 32]
b: !core/ndarray-1.0.0
  source: 1
  datatype: float64
  byteorder: little
  shape: [64, 64]
...
```

```
BLOCK 0:
    compression: bzp2
    allocated_size: 8321
    used_size: 8321
    data_size: 8192
    data: b'9f067d0981bbe13fba7163dfcb9be43f168e4662...'
```

```
BLOCK 1:
    compression: bzp2
    allocated_size: 31752
    used_size: 31752
    data_size: 32768
    data: b'48dcff21d842c73fcabaa415813eda3f08c9d228...'
```

```
#ASDF BLOCK INDEX
%YAML 1.1
---
- 631
- 9006
...
```

The lz4 compression algorithm is also supported, but requires the optional lz4 package in order to work.

When reading a file with compressed blocks, the blocks will be automatically decompressed when accessed. If a file with compressed blocks is read and then written out again, by default the new file will use the same compression as the original file. This behavior can be overridden by explicitly providing a different compression algorithm when writing the file out again.

```
import asdf

# Open a file with some compression
af = asdf.open('compressed.asdf')

# Use the same compression when writing out a new file
af.write_to('same.asdf')

# Or specify the (possibly different) algorithm to use when writing out
af.write_to('different.asdf', all_array_compression='lz4')
```

3.2.7 Memory mapping

By default, all internal array data is memory mapped using `numpy.memmap`. This allows for the efficient use of memory even when reading files with very large arrays. The use of memory mapping means that the following usage pattern is not permitted:

```
import asdf

with asdf.open('my_data.asdf') as af:
    ...

af.tree
```

Specifically, if an ASDF file has been opened using a `with` context, it is not possible to access the file contents outside of the scope of that context, because any memory mapped arrays will no longer be available.

It may sometimes be useful to copy array data into memory instead of using memory maps. This can be controlled by passing the `copy_arrays` parameter to either the `AsdfFile` constructor or `asdf.open`. By default, `copy_arrays=False`.

3.3 Using extensions

According to Wikipedia, serialization "is the process of translating data structures or object state into a format that can be stored…and reconstructed later"[1].

The power of ASDF is that it provides the ability to store, or serialize, the state of Python objects into a *human-readable* data format. The state of those objects can later be restored by another program in a process called deserialization.

While ASDF is capable of serializing basic Python types and Numpy arrays out of the box, it can also be extended to serialize arbitrary custom data types. This section discusses the extension mechanism from a user's perspective. For documentation on creating extensions, see *Extensions*.

Even though this particular implementation of ASDF necessarily serializes Python data types, in theory an ASDF implementation in another language could read the resulting file and reconstruct an analogous type in that language. Conversely, this implementation can read ASDF files that were written by other implementations of ASDF as long as the proper extensions are available.

3.3.1 The built-in extension

The ability to serialize the following types is provided by `asdf`'s built-in extension:

- `dict`
- `list`
- `str`
- `int`
- `float`
- `complex`
- `numpy.ndarray`

The built-in extension is packaged with `asdf` and is automatically used when reading and writing files. Users can not control the use of the built-in extension and in general they need not concern themselves with the details of its implementation. However, it is useful to be aware that the built-in extension is always in effect when reading and writing ASDF files.

3.3.2 Custom types

For the purposes of this documentation, a "custom type" is any data type that can not be serialized by the built-in extension.

In order for a particular custom type to be serialized, a special class called a "converter" must be implemented. Each converter defines how the corresponding custom type will be serialized and deserialized. More details on how converters are implemented can be found in *Converters*. Users should never have to refer to converter implementations directly; they simply enable `asdf` to recognize and process custom types.

In addition to converters, each custom type may have a corresponding schema, which is used for validation. The definition of the schema if present is closely tied to the definition of the converter. More details on schema validation can be found in *Schema validation*.

Schemas are generally versioned and change in sync with their associated converters. The version number will increase whenever a schema (and therefore the converter implementation) changes.

[1] https://en.wikipedia.org/wiki/Serialization

3.3.3 Extensions

In order for the converters and schemas to be used by asdf, they must be packaged into an **extension** class. In general, the details of extensions are irrelevant to users of asdf. However, users need to be aware of extensions in the following two scenarios:

- when storing custom data types to files to be written
- when reading files that contain custom data types

These scenarios require the use of custom extensions (the built-in extension is always used). There are two ways to use custom extensions, which are detailed below in *Extensions from other packages* and *Explicit use of extensions*.

Writing custom types to files

asdf is not capable of serializing any custom type unless an extension is provided that defines how to serialize that type. Attempting to do so will cause an error when trying to write the file. For details on developing support for custom types and extensions, see *Extensions*.

Reading files with custom types

The asdf software is capable of reading files that contain custom data types even if the extension that was used to create the file is not present. However, the extension is required in order to properly deserialize the original type.

If the necessary extension is **not** present, the custom data types will simply appear in the tree as a nested combination of basic data types. The structure of this data will mirror the structure of the YAML objects in the ASDF file.

In this case, a warning will occur by default to indicate to the user that the custom type in the file was not recognized and can not be deserialized. To suppress these warnings, users should pass `ignore_unrecognized_tag=True` to `asdf.open`.

Even if an extension for the custom type is present, it does not guarantee that the type can be deserialized successfully. Instantiating the custom type may involve additional software dependencies, which, if not present, will cause an error when the type is deserialized. Users should be aware of the dependencies that are required for instantiating custom types when reading ASDF files.

3.3.4 Custom types, extensions, and versioning

Tags and schemas that follow best practices are versioned. This allows changes to tags and schemas to be recorded, and it allows asdf to define behavior with respect to version compatibility.

Tag and schema versions may change for several reasons. One common reason is to reflect a change to the API of the custom type that a tag represents. This typically corresponds to an update to the version of the software that defines that custom type.

Since ASDF is designed to be an archival file format, extension authors are encouraged to maintain backwards compatibility with all older tag versions.

Reading files

When `asdf` encounters a tagged object in a file, it will compare the URI of the tag in the file with the list of tags handled by available converters. The first matching converter will be selected to deserialize the object. If no such converters exist, the library will emit a warning and the object will be presented to the user in its primitive form.

If multiple converters are present that both handle the same tag, the first found by the library will be used. Users may disable a converter by removing its extension with the `remove_extension` method.

Writing files

When writing a object to a file, `asdf` compares the object's type to the list of types handled by available converters. The first matching converter will be selected to serialize the object. If no such converters exist, the library will raise an error.

If multiple converters are present that both handle the same type, the first found by the library will be used. Users may disable a converter by removing its extension with the `remove_extension` method.

3.3.5 Extensions from other packages

Some external packages may define extensions that allow `asdf` to recognize some or all of the types that are defined by that package. Such packages may install the extension class as part of the package itself (details for developers can be found in *Installing extensions via entry points*).

If the package installs its extension, then `asdf` will automatically detect the extension and use it when processing any files. No specific action is required by the user in order to successfully read and write custom types defined by the extension for that particular package.

Users can use the `extensions` command of the `asdftool` command line tool in order to determine which packages in the current Python environment have installed ASDF extensions:

```
$ asdftool extensions -s
Extension Name: 'bizbaz' (from bizbaz 1.2.3) Class: bizbaz.io.asdf.extension.BizbazExtension
Extension Name: 'builtin' (from asdf 2.0.0) Class: asdf.extension.BuiltinExtension
```

The output will always include the built-in extension, but may also display other extensions from other packages, depending on what is installed.

3.3.6 Explicit use of extensions

Sometimes no packaged extensions are provided for the types you wish to serialize. In this case, it is necessary to explicitly install any necessary extension classes when reading and writing files that contain custom types.

The config object returned from `asdf.get_config` offers an `add_extension` method that can be used to install an extension for the remainder of the current Python session.

Consider the following example where there exists a custom type `MyCustomType` that needs to be written to a file. An extension is defined `MyCustomExtension` that contains a converter that can serialize and deserialize `MyCustomType`. Since `MyCustomExtension` is not installed by any package, we will need to manually install it:

```
import asdf

...
```

(continues on next page)

(continued from previous page)

```
asdf.get_config().add_extension(MyCustomExtension())
af = asdf.AsdfFile()
af.tree = {'thing': MyCustomType('foo') }
# This call would cause an error if the proper extension was not
# provided to the constructor
af.write_to('custom.asdf')
```

Note that the extension class must actually be instantiated when it is passed to `add_extension`.

To read the file (in a new session) we again need to install the extension first:

```
import asdf

asdf.get_config().add_extension(MyCustomExtension())
af = asdf.open('custom.asdf')
```

3.3.7 Extension checking

When writing ASDF files using this software, metadata about the extensions that were used to create the file will be added to the file itself. This includes the extension's URI, which uniquely identifies a particular version of the extension.

When reading files with extension metadata, `asdf` can check whether the required extensions are present before processing the file. If a required extension is not present, `asdf` will issue a warning.

It is possible to turn these warnings into errors by using the `strict_extension_check` parameter of `asdf.open`. If this parameter is set to `True`, then opening the file will fail if any of the required extensions are missing.

3.4 Schema validation

Schema validation is used to determine whether an ASDF file is well formed. All ASDF files must conform to the schemas defined by the ASDF Standard. Schema validation occurs when reading ASDF files (using `asdf.open`), and also when writing them out (using `AsdfFile.write_to` or `AsdfFile.update`).

Schema validation also plays a role when using custom extensions (see *Using extensions* and *Extensions*). Extensions must provide schemas for the types that they serialize. When writing a file with custom types, the output is validated against the schemas corresponding to those types. If the appropriate extension is installed when reading a file with custom types, then the types will be validated against the schemas provided by the corresponding extension.

3.4.1 Custom schemas

Every ASDF file is validated against the ASDF Standard, and also against any schemas provided by custom extensions. However, it is sometimes useful for particular applications to impose additional restrictions when deciding whether a given file is valid or not.

For example, consider an application that processes digital image data. The application expects the file to contain an image, and also some metadata about how the image was created. The following example schema reflects these expectations:

```
%YAML 1.1
---
id: "http://example.com/schemas/your-custom-schema"
```

(continues on next page)

(continued from previous page)

```
$schema: "http://stsci.edu/schemas/yaml-schema/draft-01"

type: object
properties:
  image:
    description: An ndarray containing image data.
    $ref: "ndarray-1.0.0"

  metadata:
    type: object
    description: Metadata about the image
    properties:
      time:
        description: |
          A timestamp for when the image was created, in UTC.
        type: string
        format: date-time
      resolution:
        description: |
          A 2D array representing the resolution of the image (N x M).
        type: array
        items:
          type: integer
          number: 2

required: [image, metadata]
additionalProperties: true
```

This schema restricts the kinds of files that will be accepted as valid to those that contain a top-level `image` property that is an `ndarray`, and a top-level `metadata` property that contains information about the time the image was taken and the resolution of the image.

In order to use this schema for a secondary validation pass, we pass the `custom_schema` argument to either `asdf.open` or the `AsdfFile` constructor. Assume that the schema file lives in `image_schema.yaml`, and we wish to open a file called `image.asdf`. We would open the file with the following code:

```
import asdf
af = asdf.open('image.asdf', custom_schema='image_schema.yaml')
```

Similarly, if we wished to use this schema when creating new files:

```
new_af = asdf.AsdfFile(custom_schema='image_schema.yaml')
...
```

If your custom schema is registered with ASDF in an extension, you may pass the schema URI (`http://example.com/schemas/your-custom-schema`, in this case) instead of a file path.

Note: The top-level core schemas can be found here.

3.5 Versioning and Compatibility

There are several different versions to keep in mind when discussing ASDF:

- The software package version
- The ASDF Standard version
- The ASDF file format version
- Individual tag, schema, and extension versions

Each ASDF file contains information about the various versions that were used to create the file. The most important of these are the ASDF Standard version and the ASDF file format version. A particular version of the ASDF software package will explicitly provide support for specific combinations of these versions.

Tag, schema, and extension versions are also important for serializing and deserializing data types that are stored in ASDF files. A detailed discussion of these versions from a user perspective can be found in *Custom types, extensions, and versioning*.

Since ASDF is designed to serve as an archival format, this library is careful to maintain backwards compatibility with older versions of the ASDF Standard, ASDF file format, and core tags. However, since deserializing custom tags requires other software packages, backwards compatibility is often contingent on the available versions of such software packages.

In general, forward compatibility with newer versions of the ASDF Standard and ASDF file format is not supported by the software.

When creating new ASDF files, it is possible to control the version of the file format that is used. This can be specified by passing the `version` argument to either the `AsdfFile` constructor when the file object is created, or to the `AsdfFile.write_to` method when it is written. By default, the latest version of the file format will be used. Note that this option has no effect on the versions of tags from custom extensions.

3.6 External References

3.6.1 Tree References

ASDF files may reference items in the tree in other ASDF files. The syntax used in the file for this is called "JSON Pointer", but users of `asdf` can largely ignore that.

First, we'll create a ASDF file with a couple of arrays in it:

```
import asdf
from asdf import AsdfFile
import numpy as np

tree = {
    'a': np.arange(0, 10),
    'b': np.arange(10, 20)
}

target = AsdfFile(tree)
target.write_to('target.asdf')
```

target.asdf

```
#ASDF 1.0.0
#ASDF_STANDARD 1.5.0
%YAML 1.1
%TAG ! tag:stsci.edu:asdf/
--- !core/asdf-1.1.0
asdf_library: !core/software-1.0.0 {author: The ASDF Developers, homepage: 'http://github.
↪com/asdf-format/asdf',
  name: asdf, version: 2.12.1.dev90+g784740b}
history:
  extensions:
  - !core/extension_metadata-1.0.0
    extension_class: asdf.extension.BuiltinExtension
    software: !core/software-1.0.0 {name: asdf, version: 2.12.1.dev90+g784740b}
a: !core/ndarray-1.0.0
  source: 0
  datatype: int64
  byteorder: little
  shape: [10]
b: !core/ndarray-1.0.0
  source: 1
  datatype: int64
  byteorder: little
  shape: [10]
...
```

```
BLOCK 0:
    allocated_size: 80
    used_size: 80
    data_size: 80
    data: b'00000000000000001000000000000002000000...'
```

```
BLOCK 1:
    allocated_size: 80
    used_size: 80
    data_size: 80
    data: b'0a000000000000000b000000000000000c000000...'
```

```
#ASDF BLOCK INDEX
%YAML 1.1
---
- 619
- 753
...
```

Then we will reference those arrays in a couple of different ways. First, we'll load the source file in Python and use the `make_reference` method to generate a reference to array `a`. Second, we'll work at the lower level by manually writing a JSON Pointer to array `b`, which doesn't require loading or having access to the target file.

```
ff = AsdfFile()

with asdf.open('target.asdf') as target:
```

(continues on next page)

(continued from previous page)

```
    ff.tree['my_ref_a'] = target.make_reference(['a'])

ff.tree['my_ref_b'] = {'$ref': 'target.asdf#b'}

ff.write_to('source.asdf')
```

source.asdf

```
#ASDF 1.0.0
#ASDF_STANDARD 1.5.0
%YAML 1.1
%TAG ! tag:stsci.edu:asdf/
--- !core/asdf-1.1.0
asdf_library: !core/software-1.0.0 {author: The ASDF Developers, homepage: 'http://github.
↪com/asdf-format/asdf',
  name: asdf, version: 2.12.1.dev90+g784740b}
history:
  extensions:
  - !core/extension_metadata-1.0.0
    extension_class: asdf.extension.BuiltinExtension
    software: !core/software-1.0.0 {name: asdf, version: 2.12.1.dev90+g784740b}
my_ref_a: {$ref: target.asdf#a}
my_ref_b: {$ref: target.asdf#b}
...
```

Calling `find_references` will look up all of the references so they can be used as if they were local to the tree. It doesn't actually move any of the data, and keeps the references as references.

```
with asdf.open('source.asdf') as ff:
    ff.find_references()
    assert ff.tree['my_ref_b'].shape == (10,)
```

On the other hand, calling `resolve_references` places all of the referenced content directly in the tree, so when we write it out again, all of the external references are gone, with the literal content in its place.

```
with asdf.open('source.asdf') as ff:
    ff.resolve_references()
    ff.write_to('resolved.asdf')
```

resolved.asdf

```
#ASDF 1.0.0
#ASDF_STANDARD 1.5.0
%YAML 1.1
%TAG ! tag:stsci.edu:asdf/
--- !core/asdf-1.1.0
asdf_library: !core/software-1.0.0 {author: The ASDF Developers, homepage: 'http://github.
↪com/asdf-format/asdf',
  name: asdf, version: 2.12.1.dev90+g784740b}
history:
```

(continues on next page)

(continued from previous page)

```
  extensions:
  - !core/extension_metadata-1.0.0
    extension_class: asdf.extension.BuiltinExtension
    software: !core/software-1.0.0 {name: asdf, version: 2.12.1.dev90+g784740b}
my_ref_a: !core/ndarray-1.0.0
  source: 0
  datatype: int64
  byteorder: little
  shape: [10]
my_ref_b: !core/ndarray-1.0.0
  source: 1
  datatype: int64
  byteorder: little
  shape: [10]
...
```

```
BLOCK 0:
    allocated_size: 80
    used_size: 80
    data_size: 80
    data: b'00000000000000001000000000000002000000...'
```

```
BLOCK 1:
    allocated_size: 80
    used_size: 80
    data_size: 80
    data: b'0a000000000000000b000000000000000c000000...'
```

```
#ASDF BLOCK INDEX
%YAML 1.1
---
- 633
- 767
...
```

A similar feature provided by YAML, anchors and aliases, also provides a way to support references within the same file. These are supported by asdf, however the JSON Pointer approach is generally favored because:

- It is possible to reference elements in another file
- Elements are referenced by location in the tree, not an identifier, therefore, everything can be referenced.

Anchors and aliases are handled automatically by asdf when the data structure is recursive. For example here is a dictionary that is included twice in the same tree:

```
d = {'foo': 'bar'}
d['baz'] = d
tree = {'d': d}

ff = AsdfFile(tree)
ff.write_to('anchors.asdf')
```

anchors.asdf

```
#ASDF 1.0.0
#ASDF_STANDARD 1.5.0
%YAML 1.1
%TAG ! tag:stsci.edu:asdf/
--- !core/asdf-1.1.0
asdf_library: !core/software-1.0.0 {author: The ASDF Developers, homepage: 'http://github.
↪com/asdf-format/asdf',
  name: asdf, version: 2.12.1.dev90+g784740b}
history:
  extensions:
  - !core/extension_metadata-1.0.0
    extension_class: asdf.extension.BuiltinExtension
    software: !core/software-1.0.0 {name: asdf, version: 2.12.1.dev90+g784740b}
d: &id001
  baz: *id001
  foo: bar
...
```

3.6.2 Array References

ASDF files can refer to array data that is stored in other files using the `ExternalArrayReference` type.

External files need not be ASDF files: ASDF is completely agnostic as to the format of the external file. The ASDF external array reference does not define how the external data file will be resolved; in fact it does not even check for the existence of the external file. It simply provides a way for ASDF files to refer to arrays that exist in external files.

Creating an external array reference is simple. Only four pieces of information are required:

- The name of the external file. Since ASDF does not itself resolve the file or check for its existence, the format of the name is not important. In most cases the name will be a path relative to the ASDF file itself, or a URI for a network resource.
- The data type of the array data. This is a string representing any valid `numpy.dtype`.
- The shape of the data array. This is a tuple representing the dimensions of the array data.
- The array data `target`. This is either an integer or a string that indicates to the user something about how the data array should be accessed in the external file. For example, if there are multiple data arrays in the external file, the `target` might be an integer index. Or if the external file is an ASDF file, the `target` might be a string indicating the key to use in the external file's tree. The value and format of the `target` field is completely arbitrary since ASDF will not use it itself.

As an example, we will create a reference to an external CSV file. We will assume that one of the rows of the CSV file contains the array data we care about:

```
import asdf

csv_data_row = 10 # The row of the CSV file containing the data we want
csv_row_size = 100 # The size of the array
extref = asdf.ExternalArrayReference('data.csv', csv_data_row, "int64", (csv_row_size,))

tree = {'csv_data': extref}
```

(continues on next page)

(continued from previous page)

```
af = asdf.AsdfFile(tree)
af.write_to('external_array.asdf')
```

external_array.asdf

```
#ASDF 1.0.0
#ASDF_STANDARD 1.5.0
%YAML 1.1
%TAG ! tag:stsci.edu:asdf/
--- !core/asdf-1.1.0
asdf_library: !core/software-1.0.0 {author: The ASDF Developers, homepage: 'http://github.
↪com/asdf-format/asdf',
  name: asdf, version: 2.12.1.dev90+g784740b}
history:
  extensions:
  - !core/extension_metadata-1.0.0
    extension_class: asdf.extension.BuiltinExtension
    software: !core/software-1.0.0 {name: asdf, version: 2.12.1.dev90+g784740b}
csv_data: !core/externalarray-1.0.0
  datatype: int64
  fileuri: data.csv
  shape: [100]
  target: 10
...
```

When reading a file containing external references, the user is responsible for using the information in the `ExternalArrayReference` type to open the external file and retrieve the associated array data.

3.7 Saving history entries

`asdf` has a convenience method for notating the history of transformations that have been performed on a file.

Given a `AsdfFile` object, call `add_history_entry`, given a description of the change and optionally a description of the software (i.e. your software, not `asdf`) that performed the operation.

```
from asdf import AsdfFile
import numpy as np

tree = {
    'a': np.random.rand(32, 32)
}

ff = AsdfFile(tree)
ff.add_history_entry(
    "Initial random numbers",
    {'name': 'asdf examples',
     'author': 'John Q. Public',
     'homepage': 'http://github.com/asdf-format/asdf',
```

(continues on next page)

(continued from previous page)

```
    'version': '0.1'})
ff.write_to('example.asdf')
```

example.asdf

```
#ASDF 1.0.0
#ASDF_STANDARD 1.5.0
%YAML 1.1
%TAG ! tag:stsci.edu:asdf/
--- !core/asdf-1.1.0
asdf_library: !core/software-1.0.0 {author: The ASDF Developers, homepage: 'http://github.
↪com/asdf-format/asdf',
  name: asdf, version: 2.12.1.dev90+g784740b}
history:
  entries:
  - !core/history_entry-1.0.0
    description: Initial random numbers
    software: !core/software-1.0.0 {author: John Q. Public, homepage: 'http://github.com/
↪asdf-format/asdf',
      name: asdf examples, version: '0.1'}
    time: 2022-08-12 19:47:17
  extensions:
  - !core/extension_metadata-1.0.0
    extension_class: asdf.extension.BuiltinExtension
    software: !core/software-1.0.0 {name: asdf, version: 2.12.1.dev90+g784740b}
a: !core/ndarray-1.0.0
  source: 0
  datatype: float64
  byteorder: little
  shape: [32, 32]
...
```

```
BLOCK 0:
    allocated_size: 8192
    used_size: 8192
    data_size: 8192
    data: b'04e2f4b2e3f0dc3fd68b8524129de53f42ccdd6b...'
```

```
#ASDF BLOCK INDEX
%YAML 1.1
---
- 800
...
```

asdf automatically saves history metadata about the extensions that were used to create the file. This information is used when opening files to determine if the proper extensions are installed (see *Extension checking* for more details).

3.8 Saving ASDF in FITS

Note: This section is about packaging entire ASDF files inside of FITS data format files. This is probably only of interest to astronomers. Making use of this feature requires the `astropy` package to be installed.

Sometimes you may need to store the structured data supported by ASDF inside of a FITS file in order to be compatible with legacy tools that support only FITS.

First, create an `HDUList` object using `astropy.io.fits`. Here, we are building an `HDUList` from scratch, but it could also have been loaded from an existing file.

We will create a FITS file that has two image extensions, SCI and DQ respectively.

```
from astropy.io import fits

hdulist = fits.HDUList()
hdulist.append(fits.ImageHDU(np.arange(512, dtype=float), name='SCI'))
hdulist.append(fits.ImageHDU(np.arange(512, dtype=float), name='DQ'))
```

Next we make a tree structure out of the data in the FITS file. Importantly, we use the *same* array references in the FITS `HDUList` and store them in the tree. By doing this, ASDF will automatically refer to the data in the regular FITS extensions.

```
tree = {
    'model': {
        'sci': {
            'data': hdulist['SCI'].data,
        },
        'dq': {
            'data': hdulist['DQ'].data,
        }
    }
}
```

Now we take both the FITS `HDUList` and the ASDF tree and create an `AsdfInFits` object.

```
from asdf import fits_embed

ff = fits_embed.AsdfInFits(hdulist, tree)
ff.write_to('embedded_asdf.fits', overwrite=True)
```

The special ASDF extension in the resulting FITS file contains the following data. Note that the data source of the arrays uses the `fits:` prefix to indicate that the data comes from a FITS extension:

content.asdf

```
#ASDF 1.0.0
#ASDF_STANDARD 1.5.0
%YAML 1.1
%TAG ! tag:stsci.edu:asdf/
--- !core/asdf-1.1.0
asdf_library: !core/software-1.0.0 {author: The ASDF Developers, homepage: 'http://github.
```

(continues on next page)

(continued from previous page)

```
↪com/asdf-format/asdf',
  name: asdf, version: 2.12.1.dev90+g784740b}
history:
  extensions:
  - !core/extension_metadata-1.0.0
    extension_class: asdf.extension.BuiltinExtension
    software: !core/software-1.0.0 {name: asdf, version: 2.12.1.dev90+g784740b}
model:
  dq:
    data: !core/ndarray-1.0.0
      source: fits:DQ,1
      datatype: float64
      byteorder: big
      shape: [512]
  sci:
    data: !core/ndarray-1.0.0
      source: fits:SCI,1
      datatype: float64
      byteorder: big
      shape: [512]
...
```

To load an ASDF-in-FITS file, simply open it using `asdf.open`. The returned value will be an `AsdfInFits` object, which can be used in the same way as any other `AsdfFile` object.

```
with asdf.open('embedded_asdf.fits') as asdf_in_fits:
    science = asdf_in_fits.tree['model']['sci']
```

3.9 Rendering ASDF trees

The `asdf.info` function prints a representation of an ASDF tree to stdout. For example:

```
>>> asdf.info('path/to/some/file.asdf')
root (AsdfObject)
├─asdf_library (Software)
│ ├─author (str): The ASDF Developers
│ ├─homepage (str): http://github.com/asdf-format/asdf
│ ├─name (str): asdf
│ └─version (str): 2.5.1
├─history (dict)
│ └─extensions (list) ...
└─data (dict)
  └─example_key (str): example value
```

The first argument may be a `str` or `pathlib.Path` filesystem path, or an `AsdfFile` or sub-node of an ASDF tree.

By default, `asdf.info` limits the number of lines, and line length, of the displayed tree. The `max_rows` parameter controls the number of lines, and `max_cols` controls the line length. Set either to `None` to disable that limit.

An integer `max_rows` will be interpreted as an overall limit on the number of displayed lines. If `max_rows` is a tuple, then each member limits lines per node at the depth corresponding to its tuple index. For example, to show all top-level

nodes and 5 of each's children:

If the attribute is described in a schema, the info functionality will see if it has an associated title and if it does, display it as a comment on the same line. This provides a way for users to see more information about the the attribute in a similar way that FITS header comments are used.

```
>>> asdf.info('file.asdf', max_rows=(None, 5))
...
```

The `AsdfFile.info` method behaves similarly to `asdf.info`, rendering the tree of the associated `AsdfFile`.

Normally `asdf.info` will not show the contents of asdf nodes turned into Python custom objects, but if that object supports a special method, you may see the contents of such objects. See *Making converted object's contents visible to info and search* for how to implement such support for `asdf.info` and `asdf.search`.

3.10 Searching the ASDF tree

The `AsdfFile` search interface provides a way to interactively discover the locations and values of nodes within the ASDF tree. We can search for nodes by key/index, type, or value.

3.10.1 Basic usage

Initiate a search by calling `AsdfFile.search` on an open file:

```
>>> af.search()
root (AsdfObject)
├─asdf_library (Software)
│ ├─author (str): The ASDF Developers
│ ├─homepage (str): http://github.com/asdf-format/asdf
│ ├─name (str): asdf
│ └─version (str): 2.5.1
├─history (dict)
│ └─extensions (list) ...
└─data (dict)
  └─example_key (str): example value

>>> af.search('example')
root (AsdfObject)
└─data (dict)
  └─example_key (str): example value
```

The search returns an `AsdfSearchResult` object that displays in the Python console as a rendered tree. For single-node search results, the `AsdfSearchResult.path` property contains the Python code required to reference that node directly:

```
>>> af.search('example').path
"root['data']['example_key']"
```

While the `AsdfSearchResult.node` property contains the actual value of the node:

```
>>> af.search('example').node
'example value'
```

For searches with multiple matching nodes, use the `AsdfSearchResult.paths` and `AsdfSearchResult.nodes` properties instead:

```
>>> af.search('duplicate_key').paths
["root['data']['duplicate_key']", "root['other_data']['duplicate_key']"]
>>> af.search('duplicate_key').nodes
["value 1", "value 2"]
```

To replace matching nodes with a new value, use the `AsdfSearchResult.replace` method:

```
>>> af.search('example').replace('replacement value')
>>> af.search('example').node
'replacement value'
```

The first argument to `AsdfFile.search` searches by dict key or list/tuple index. We can also search by type, value, or any combination thereof:

```
>>> af.search('foo') # Find nodes with key containing the string 'foo'
...
>>> af.search(type=int) # Find nodes that are instances of int
...
>>> af.search(value=10) # Find nodes whose value is equal to 10
...
>>> af.search('foo', type=int, value=10) # Find the intersection of the above
```

3.10.2 Chaining searches

The return value of `AsdfFile.search`, `asdf.search.AsdfSearchResult`, has its own search method, so it's possible to chain searches together. This is useful when you need to see intermediate results before deciding how to further narrow the search.

```
>>> af.search() # See an overview of the entire ASDF tree
...
>>> af.search().search(type='NDArrayType') # Find only ndarrays
...
>>> af.search().search(type='NDArrayType').search('err') # Only ndarrays with 'err' in the␣
↪key
```

3.10.3 Descending into child nodes

Another way to narrow the search is to use the index operator to descend into a child node of the current tree root:

```
>>> af.search()['data'] # Restrict search to the 'data' child
...
>>> af.search()['data'].search(type=int) # Find integer descendants of 'data'
```

3.10.4 Regular expression searches

Any string argument to search is interpeted as a regular expression. For example, we can search for nodes whose keys start with a particular string:

```
>>> af.search('foo') # Find nodes with 'foo' anywhere in the key
...
>>> af.search('^foo') # Find only nodes whose keys start with 'foo'
...
```

Note that all node keys (even list indices) will be converted to string before the regular expression is matched:

```
>>> af.search('^7$') # Returns all nodes with key '7' or index 7
...
```

When the `type` argument is a string, the search compares against the fully-qualified class name of each node:

```
>>> af.search(type='asdf.tags.core.Software') # Find instances of ASDF's Software type
...
>>> af.search(type='^asdf\.') # Find all ASDF objects
...
```

When the `value` argument is a string, the search compares against the string representation of each node's value.

```
>>> af.search(value='^[0-9]{4}-[0-9]{2}-[0-9]{2}$') # Find values that look like dates
...
```

3.10.5 Arbitrary search criteria

If `key`, `type`, and `value` aren't sufficient, we can also provide a callback function to search by arbitrary criteria. The `filter` parameter accepts a callable that receives the node under consideration, and returns `True` to keep it or `False` to reject it from the search results. For example, to search for NDArrayType with a particular shape:

```
>>> af.search(type='NDArrayType', filter=lambda n: n.shape[0] == 1024)
...
```

3.10.6 Formatting search results

The `AsdfSearchResult` object displays its content as a rendered tree with reasonable defaults for maximum number of lines and columns displayed. To change those values, we call `AsdfSearchResult.format`:

```
>>> af.search(type=float) # Displays limited rows
...
>>> af.search(type=float).format(max_rows=None) # Show all matching rows
...
```

Like `AsdfSearchResult.search`, calls to format may be chained:

```
>>> af.search('time').format(max_rows=10).search(type=str).format(max_rows=None)
...
```

CHAPTER

FOUR

CONFIGURATION

Version 2.8 of this library introduced a new mechanism, `AsdfConfig`, for setting global configuration options. Currently available options are limited, but we expect to eventually move many of the `AsdfFile.__init__` and `AsdfFile.write_to` keyword arguments to `AsdfConfig`.

4.1 AsdfConfig and you

The `AsdfConfig` class provides properties that can be adjusted to change the behavior of the `asdf` library for all files. For example, to disable schema validation on read:

```
>>> import asdf
>>> asdf.get_config().validate_on_read = False
```

This will prevent validation on any subsequent call to `open`.

4.1.1 Obtaining an AsdfConfig instance

There are two methods available that give access to an `AsdfConfig` instance: `get_config` and `config_context`. The former simply returns the currently active config:

```
>>> import asdf
>>> asdf.get_config()
<AsdfConfig
  array_inline_threshold: None
  default_version: 1.5.0
  io_block_size: -1
  legacy_fill_schema_defaults: True
  validate_on_read: True
>
```

The latter method, `config_context`, returns a context manager that yields a copy of the currently active config. The copy is also returned by subsequent calls to `get_config`, but only until the context manager exits. This allows for short-lived configuration changes that do not impact other code:

```
>>> import asdf
>>> with asdf.config_context() as config:
...   config.validate_on_read = False
...   asdf.get_config()
...
```

(continues on next page)

(continued from previous page)

```
<AsdfConfig
  array_inline_threshold: None
  default_version: 1.5.0
  io_block_size: -1
  legacy_fill_schema_defaults: True
  validate_on_read: False
>
>>> asdf.get_config()
<AsdfConfig
  array_inline_threshold: None
  default_version: 1.5.0
  io_block_size: -1
  legacy_fill_schema_defaults: True
  validate_on_read: True
>
```

4.1.2 Special note to library maintainers

Libraries that use `asdf` are encouraged to only modify `AsdfConfig` within a surrounding call to `config_context`. The downstream library will then be able to customize `asdf`'s behavior without impacting other libraries or clobbering changes made by the user.

4.2 Config options

4.2.1 array_inline_threshold

The threshold number of array elements under which arrays are automatically stored inline in the ASDF tree instead of in binary blocks. If `None`, array storage type is not managed automatically.

Defaults to `None`.

4.2.2 default_version

The default ASDF Standard version used for new files. This can be overridden on an individual file basis (using the version argument to `AsdfFile.__init__`) or set here to change the default for all new files created in the current session.

Defaults to the latest stable ASDF Standard version.

4.2.3 io_block_size

The buffer size used when reading and writing to the filesystem. Users may wish to adjust this value to improve I/O performance. Set to -1 to use the preferred block size for each file, as reported by st_blksize.

Defaults to -1.

4.2.4 legacy_fill_schema_defaults

Flag that controls filling default values from schemas for older versions of the ASDF Standard. This library used to remove nodes from the tree whose values matched the default property in the schema. That behavior was changed in asdf 2.8, but in order to read files produced by older versions of the library, default values must still be filled from the schema for ASDF Standard <= 1.5.0.

Set to False to disable filling default values from the schema for these older ASDF Standard versions. The flag has no effect for ASDF Standard >= 1.6.0.

Defaults to True.

4.2.5 validate_on_read

Flag that controls schema validation of the ASDF tree when opening files. Users who trust the source of their files may wish to disable validation on read to improve performance.

Defaults to True.

4.3 Additional AsdfConfig features

AsdfConfig also provides methods for adding and removing plugins at runtime. For example, the AsdfConfig.add_resource_mapping method can be used to register a schema, which can then be used to validate a file:

```
>>> import asdf
>>> content = b"""
... %YAML 1.1
... ---
... $schema: http://stsci.edu/schemas/yaml-schema/draft-01
... id: http://example.com/example-project/schemas/foo-1.0.0
... type: object
... properties:
...   foo:
...   type: string
... required: [foo]
... ...
... """
>>> asdf.get_config().add_resource_mapping({"http://example.com/example-project/schemas/foo-
↪1.0.0": content})
>>> af = asdf.AsdfFile(custom_schema="http://example.com/example-project/schemas/foo-1.0.0")
>>> af.validate()
Traceback (most recent call last):
...
jsonschema.exceptions.ValidationError: 'foo' is a required property
...
>>> af["foo"] = "bar"
>>> af.validate()
```

See the AsdfConfig API documentation for more detail.

CHAPTER

FIVE

COMMAND LINE TOOL

asdf includes a command-line tool, `asdftool` that performs a number of useful operations:

- `explode`: Convert a self-contained ASDF file into exploded form (see *Saving external arrays*).
- `implode`: Convert an ASDF file in exploded form into a self-contained file.
- `defragment`: Remove unused blocks and extra space.
- `diff`: Report differences between two ASDF files.
- `edit`: Edit the YAML portion of an ASDF file.
- `remove-hdu`: Remove ASDF extension from ASDF-in-FITS file (requires `astropy`, see *Saving ASDF in FITS*).
- `info`: Print a rendering of an ASDF tree.
- `extensions`: Show information about installed extensions (see *Extensions from other packages*).
- `tags`: List currently available tags.
- `to_yaml`: Inline all of the data in an ASDF file so that it is pure YAML.

Run `asdftool --help` for more information.

Part II

Extending ASDF

CHAPTER

SIX

COMMON USE CASES

This section is intended as a kind of index to the rest of the "Extending ASDF" documentation. Here we list common use cases and link to the relevant documentation sections that are needed to get the job done.

6.1 Validate an ASDF tree against a schema

The `asdf` library already validates individual tagged objects within the tree, but what if we want to validate the structure of the tree itself? Such "document schemas" can be associated with an `AsdfFile` using the `custom_schema` argument, but this argument accepts a URI and the asdf library needs to know how to access the schema content associated with that URI.

1. Designate a URI for the schema. See *Schemas* for recommendations on schema URI structure.
2. Write the schema. See *ASDF schemas* if you're new to authoring schemas.
3. Install the schema as an `asdf` library resource. See *Resources and resource mappings* for an overview of resources in `asdf` and options for installing them.

6.2 Serialize a new type

This section summarizes the steps needed to serialize a new type to an ASDF file. We'll describe three options, starting with the most expedient and growing progressively more formal.

6.2.1 Quick and dirty, for personal use

In this scenario, we want to serialize a new Python type to an ASDF file, but we're not planning on widely sharing the file, so we want to cut as many corners as possible. Here are the minimal steps needed to get instances of that type into the file and back again:

1. Identify the Python type to serialize. We'll need to know the fully-qualified name of the type (module path + class name).
2. Select a tag URI that will signify the type in YAML. See *Tags* for recommendations on tag URI structure.
3. Implement a `Converter` class that converts the type to YAML-serializable objects and back again. See *Converters* for a discussion of the Converter interface.
4. Implement an `Extension` class which is the vehicle for plugging our converter into the asdf library. See *Extensions* for a discussion of the Extension interface.
5. Install the extension. There are multiple ways to do this, but the path of least resistance is to install the extension at runtime using `AsdfConfig`. See *Installing extensions via AsdfConfig*.

Now instances of our type can be added to an `AsdfFile`'s tree and serialized to an ASDF file.

6.2.2 For sharing with other Python users

Now say our files are getting out into the world and into the hands of other Python users. We'll want to build an installable package around our code and use the `asdf` library's entry points to make our extension more convenient to use. We should also think about adding a schema that validates our tagged objects, so if someone manually edits a file and makes a mistake, we get a clear error when `asdf` opens the file.

1. Identify the Python type to serialize. We'll need to know the fully-qualified name of the type (module path + class name).
2. Select a tag URI that will signify the type in YAML. See *Tags* for recommendations on tag URI structure.
3. Designate a URI for the schema. See *Schemas* for recommendations on schema URI structure.
4. Write the schema that will validate the tagged object. See *ASDF schemas* if you're new to authoring schemas.
5. Make the schema installable as an `asdf` library resource. See *Resources and resource mappings* for an overview of resources in `asdf` and *Installing resources via entry points* for information on installing resources via an entry point.
6. Implement a `Converter` class that converts the type to YAML-serializable objects and back again. See *Converters* for a discussion of the Converter interface. Refer to the schema to ensure that the Converter is writing YAML objects correctly.
7. Implement an `Extension` class which is the vehicle for plugging our converter into the `asdf` library. See *Extensions* for a discussion of the Extension interface. We'll need to associate the schema URI with the tag URI in our tag's `TagDefinition` object.
8. Install the extension via an entry point. See *Installing extensions via entry points*.

Now anyone who installs the package containing the entry points will be able to read, write, and validate ASDF files containing our new tag!

6.2.3 For sharing with users of other languages

Finally, let's consider the case where we want to serialize instances of our type to an ASDF file that will be read using ASDF libraries written in other languages. The problem with our previous efforts is that the extension definition exists only as Python code, so here we'll want to create an additional YAML document called an extension manifest that defines the extension in a language-independent way.

1. Identify the Python type to serialize. We'll need to know the fully-qualified name of the type (module path + class name).
2. Select a tag URI that will signify the type in YAML. See *Tags* for recommendations on tag URI structure.
3. Designate a URI for the schema. See *Schemas* for recommendations on schema URI structure.
4. Write the schema that will validate the tagged object. See *ASDF schemas* if you're new to authoring schemas.
5. Write an extension manifest document that describes the tag and schema that we're including in our extension. See *Extension manifests* for information on the manifest format.
5. Make the schema and manifest installable as `asdf` library resources. See *Resources and resource mappings* for an overview of resources in `asdf` and *Installing resources via entry points* for information on installing resources via an entry point.

6. Implement a `Converter` class that converts the type to YAML-serializable objects and back again. See *Converters* for a discussion of the Converter interface. Refer to the schema to ensure that the Converter is writing YAML objects correctly.

7. Use `asdf.extension.ManifestExtension.from_uri` to populate an extension with the Converter and information from the manifest document. See *Populating an extension from a manifest* for instructions on using ManifestExtension.

8. Install the extension via an entry point. See *Installing extensions via entry points*.

That's it! Python users should experience the same convenience, but now the manifest document is available as a reference for developers who wish to implement support for reading our tagged objects in their language of choice.

6.3 Support a new block compressor

In order to support a new compression algorithm for ASDF binary blocks, we need to implement the `Compressor` interface and install that in an extension.

1. Select a 4-byte compression code that will signify the compression algorithm.

1. Implement a `Compressor` class that associates the 4-byte code with compression and decompression methods. See *Binary block compressors* for a discussion of the Compressor interface.

2. Implement an `Extension` class which is the vehicle for plugging our compressor into the `asdf` library. See *Extensions* for a discussion of the Extension interface.

3. Install the extension via one of the two available methods. See *Installing an extension* for instructions.

Now the compression algorithm will be available for both reading and writing ASDF files. Users writing files will simply need to specify the new 4-byte compression code when making calls to `asdf.AsdfFile.set_array_compression`.

CHAPTER

SEVEN

URIS IN ASDF

The ASDF format uses **U**niform **R**esource **I**dentifiers to refer to various entities such as schemas or tags. These are string identifiers that often resemble web addresses that uniquely identify the associated entity. Here are some examples of URIs that might be encountered in an ASDF file:

- `http://stsci.edu/schemas/asdf/core/ndarray-1.0.0` (URI of the ndarray-1.0.0 schema)
- `tag:stsci.edu:asdf/core/asdf-1.1.0` (URI of the asdf-1.0.0 YAML tag)
- `asdf://example.com/schemas/foo-1.0.0` (URI of the foo-1.0.0 schema)

Each of these uses a different URI *scheme*, and each is a valid URI format in ASDF.

7.1 URI vs URL

One common point of confusion is the distinction between a URI and a URL. The two are easily conflated – for example, consider the URI of the ndarray-1.0.0 schema above.

`http://stsci.edu/schemas/asdf/core/ndarray-1.0.0`

This string looks just like the URL of a web page, but if we were to attempt to visit that location in a browser, we'd get a 404 Not Found from stsci.edu. And yet it is still a valid URI!

The similarity arises from the need for URIs to be globally unique. Since web domains are already controlled by a single organization or individual, they offer a convenient way to define URIs – just reserve some path prefix off a domain you control and dole out strings with that prefix where unique identifiers are needed. But using `http://` as a URI scheme has the downside that users expect to be able to retrieve the document contents from that address.

7.2 The asdf:// URI scheme

To counter the problem of URIs vs URLs, `asdf` 2.8 introduced support for the `asdf://` URI scheme. These URIs are constructed just like `http://` or `https://` URIs, but the ASDF-specific scheme makes clear that the content cannot be fetched from a webserver.

7.3 Entities identified by URI

The following is a complete list of entity types that are identified by URI in ASDF:

7.3.1 Schemas

Schemas are expected to include an `id` property that contains the URI that identifies them. That URI is used when referring to the schema in calls to `asdf` library functions. We recommend the following pattern for schema URIs:

`asdf://<domain>/<project>/schemas/<name>-<version>`

Where `<domain>` is some domain that you control, `<project>` collects all entities for a particular ASDF project, `<name>` is the name of the schema, and `<version>` is the schema's version number. For example:

`asdf://example.com/example-project/schemas/foo-1.2.3`

7.3.2 Tags

Tags, which annotate typed objects in an ASDF file's YAML tree, are represented as URIs. Unlike schemas, there is no resource associated with the tag; no blob of bytes exists that corresponds to the URI. Instead, the URI alone communicates the type of a YAML object. We recommend the following pattern for tag URIs:

`asdf://<domain>/<project>/tags/<name>-<version>`

Where `<domain>` is some domain that you control, `<project>` collects all entities for a particular ASDF project, `<name>` is the name of the tag, and `<version>` is the tag's version number. For example:

`asdf://example.com/example-project/tags/foo-1.2.3`

7.3.3 Manifests

Manifest documents are language-independent definitions of extensions to ASDF and include an `id` property that contains the URI that identifies them. That URI is used when referring to the manifest in calls to `asdf` library functions. We recommend the following pattern for manifest URIs:

`asdf://<domain>/<project>/manifests/<name>-<version>`

Where `<domain>` is some domain that you control, `<project>` collects all entities for a particular ASDF project, `<name>` is the name of the manifest, and `<version>` is the manifest's version number. For example:

`asdf://example.com/example-project/manifests/foo-1.2.3`

7.3.4 Extensions

Finally, extensions URIs identify extensions to the ASDF format. These URIs are included in an ASDF file's metadata to advertise the fact that additional software support (beyond a core ASDF library) is needed to properly interpret the file. Like tags, these URIs are not associated with a particular resource. We recommend the following pattern for extension URIs:

`asdf://<domain>/<project>/extensions/<name>-<version>`

Where `<domain>` is some domain that you control, `<project>` collects all entities for a particular ASDF project, `<name>` is the name of the extension, and `<version>` is the extension's version number. For example:

`asdf://example.com/example-project/extensions/foo-1.2.3`

CHAPTER

EIGHT

ASDF SCHEMAS

ASDF schemas are YAML documents that describe validations to be performed on tagged objects nested within the ASDF tree or on the tree itself. Schemas can validate the presence, datatype, and value of objects and their properties, and can be combined in different ways to facilitate reuse.

These schemas, though expressed in YAML, are structured according to the JSON Schema Draft 4 specification. The excellent Understanding JSON Schema book is a great place to start for users not already familiar with JSON Schema. Just keep in mind that the book includes coverage of later drafts of the JSON Schema spec, so certain features (constant values, conditional subschemas, etc) will not be available when writing schemas for ASDF. The book makes clear which features were introduced after Draft 4.

8.1 Anatomy of a schema

Here is an example of an ASDF schema that validates an object with a numeric value and corresponding unit:

```
%YAML 1.1
---
$schema: http://stsci.edu/schemas/yaml-schema/draft-01
id: asdf://asdf-format.org/core/schemas/quantity-2.0.0

title: Quantity object containing numeric value and unit
description: >-
  An object with a numeric value, which may be a scalar
  or an array, and associated unit.

type: object
properties:
  value:
    description: A vector of one or more values
    anyOf:
      - type: number
      - tag: tag:stsci.edu:asdf/core/ndarray-1.0.0
  unit:
    description: The unit corresponding to the values
    tag: tag:stsci.edu:asdf/unit/unit-1.0.0
  required: [value, unit]
...
```

This is similar to the quantity schema, found here, of the ASDF Standard, but has been updated to reflect current recommendations regarding schemas. Let's walk through this schema line by line.

```
%YAML 1.1
---
```

These first two lines form the header of the file. The `%YAML 1.1` indicates that we're following version 1.1 of the YAML spec. The `---` marks the start of a new YAML document.

```
$schema: http://stsci.edu/schemas/yaml-schema/draft-01
```

The `$schema` property contains the URI of the schema that validates this document. Since our document is itself a schema, the URI refers to a *metaschema*. ASDF comes with three built-in metaschemas:

- `http://json-schema.org/draft-04/schema` - The JSON Schema Draft 4 metaschema. Includes basic validators and combiners.
- `http://stsci.edu/schemas/yaml-schema/draft-01` - The YAML Schema metaschema. Includes everything in JSON Schema Draft 4, plus additional YAML-specific validators including `tag` and `propertyOrder`.
- `http://stsci.edu/schemas/asdf/asdf-schema-1.0.0` - The ASDF Schema metaschema. Includes everything in YAML Schema, plus additional ASDF-specific validators that check ndarray properties.

Our schema makes use of the `tag` validator, so we're specifying the YAML Schema URI here.

```
id: asdf://asdf-format.org/core/schemas/quantity-2.0.0
```

The `id` property contains the URI that uniquely identifies our schema. This URI is how we'll refer to the schema when using the asdf library.

```
title: Quantity object containing numeric value and unit
description: >-
  An object with a numeric value, which may be a scalar
  or an array, and associated unit.
```

Title and description are optional (but recommended) documentation properties. These properties can be placed multiple times at any level of the schema and do not have an impact on the validation process.

```
type: object
```

This line invokes the `type` validator to check the data type of the top-level value. We're asserting that the type must be a YAML mapping, which in Python is represented as a `dict`.

```
properties:
```

The `properties` validator announces that we'd like to validate certain named properties of mapping. If a property is listed here and is present in the ASDF, it will be validated accordingly.

```
  value:
    description: A vector of one or more values
```

Here we're identifying a property named `value` that we'd like to validate. The `description` is used to add some additional documentation.

```
  anyOf:
```

The `anyOf` validator is one of JSON Schema's combiners. The `value` property will be validated against each of the following subschemas, and if any validates successfully, the entire `anyOf` will be considered valid. Other available combiners are `allOf`, which requires that all subschemas validate successfully, `oneOf`, which requires that one and only one of the subschemas validates, and `not`, which requires that a single subschema does *not* validate.

```
    - type: number
```

The first subschema in the list contains a `type` validator that succeeds if the entity assigned to `value` is a numeric literal.

```
    - tag: tag:stsci.edu:asdf/core/ndarray-1.0.0
```

The second subschema contains a `tag` validator, which makes an assertion regarding the YAML tag URI of the object assigned to `value`. In this subschema we're requiring the tag of an ndarray-1.0.0 object, which is how n-dimensional arrays are represented in an ASDF tree.

The net effect of the `anyOf` combiner and its two subschemas is: validate successfully if the `value` object is either a numeric literal or an n-dimensional array.

```
  unit:
    description: The unit corresponding to the values
    tag: tag:stsci.edu:asdf/unit/unit-1.0.0
```

The `unit` property has another bit of documentation and a `tag` validator that requires it to be a unit-1.0.0 object.

```
required: [value, unit]
```

Since the `properties` validator does not require the presence of its listed properties, we need another validator to do that. The `required` validator defines a list of properties that need to be present if validation is to succeed.

```
...
```

Finally, the YAML document end indicator indicates the end of the schema.

8.2 Checking schema syntax

The `check_schema` function performs basic syntax checks on a schema and will raise an error if it discovers a problem. It does not currently accept URIs and requires that the schema already be loaded into Python objects. If the schema is already registered with the asdf library as a resource (see *Resources and resource mappings*), it can be loaded and checked like this:

```
from asdf.schema import load_schema, check_schema

schema = load_schema("asdf://example.com/example-project/schemas/foo-1.0.0")
check_schema(schema)
```

Otherwise, the schema can be loaded using pyyaml directly:

```
from asdf.schema import check_schema
import yaml

schema = yaml.safe_load(open("/path/to/foo-1.0.0.yaml").read())
check_schema(schema)
```

8.3 Testing validation

Getting a schema to validate as intended can be a tricky business, so it's helpful to test validation against some example objects as you go along. The `validate` function will validate a Python object against a schema:

```
from asdf.schema import validate
import yaml

schema = yaml.safe_load(open("/path/to/foo-1.0.0.yaml").read())
obj = {"foo": "bar"}
validate(obj, schema=schema)
```

The validate function will return successfully if the object is valid, or raise an error if not.

8.4 See also:

- JSON Schema Draft 4
- Understanding JSON Schema
- Unit Schemas

CHAPTER

NINE

RESOURCES AND RESOURCE MAPPINGS

In the terminology of this library, a "resource" is a sequence of bytes associated with a URI. Currently the two types of resources recognized by `asdf` are schemas and extension manifests. Both of these are YAML documents whose associated URI is expected to match the `id` property of the document.

A "resource mapping" is an `asdf` plugin that provides access to the content for a URI. These plugins must implement the `Mapping` interface (a simple `dict` qualifies) and map `str` URI keys to `bytes` values. Resource mappings are installed into the `asdf` library via one of two routes: the `AsdfConfig.add_resource_mapping` method or the `asdf.resource_mappings` entry point.

9.1 Installing resources via AsdfConfig

The simplest way to isntall a resource into `asdf` is to add it at runtime using the `AsdfConfig.add_resource_mapping` method. For example, the following code istalls a schema for use with the `asdf.AsdfFile` custom_schema argument:

```
import asdf

content = b"""
%YAML 1.1
---
$schema: http://stsci.edu/schemas/yaml-schema/draft-01
id: asdf://example.com/example-project/schemas/foo-1.0.0
type: object
properties:
  foo:
    type: string
required: [foo]
...
"""

asdf.get_config().add_resource_mapping({
  "asdf://example.com/example-project/schemas/foo-1.0.0": content
})
```

The schema will now be available for validating files:

```
af = asdf.AsdfFile(custom_schema="asdf://example.com/example-project/schemas/foo-1.0.0")
af.validate() # Error, "foo" is missing
```

9.2 The DirectoryResourceMapping class

But what if we don't want to store our schemas in variables in the code? Storing resources in a directory tree is a common use case, so `asdf` provides a `Mapping` implementation that reads schema content from a filesystem. This is the `DirectoryResourceMapping` class.

Consider these three schemas:

```
# foo-1.0.0.yaml
id: asdf://example.com/example-project/schemas/foo-1.0.0
# ...

# bar-2.3.4.yaml
id: asdf://example.com/example-project/nested/bar-2.3.4
# ...

# baz-8.1.1.yaml
id: asdf://example.com/example-project/nested/baz-8.1.1
# ...
```

which are arranged in the following directory structure:

```
schemas
├─ foo-1.0.0.yaml
├─ README
└─ nested
    ├─ bar-2.3.4.yaml
    └─ baz-8.1.1.yaml
```

Our goal is to install all schemas in the directory tree so that they are available for use with `asdf`. The `DirectoryResourceMapping` class can do that for us, but we need to show it how to construct the schema URIs from the file paths *without reading the id property from the files*. This requirement is a performance consideration; not all resources are used in every session, and if `asdf` were to read and parse all available files when plugins are loaded, the first call to `asdf.open` would be intolerably slow.

We should configure `DirectoryResourceMapping` like this:

```
import asdf
from asdf.resource import DirectoryResourceMapping

mapping = DirectoryResourceMapping(
    "/path/to/schemas",
    "asdf://example.com/example-project/schemas/",
    recursive=True,
    filename_pattern="*.yaml",
    stem_filename=True
)

asdf.get_config().add_resource_mapping(mapping)
```

The first argument is the path to the schemas directory on the filesystem. The second argument is the prefix that should be prepended to file paths relative to that root when constructing the schema URIs. The `recursive` argument tells the class to descend into the `nested` directory when searching for schemas, `filename_pattern` is a glob pattern chosen to exclude our README file, and `stem_filename` causes the class to drop the `.yaml` suffix when constructing URIs.

We can test that our configuration is correct by asking `asdf` to read and parse one of the schemas:

```
from asdf.schema import load_schema

uri = "asdf://example.com/example-project/schemas/nested/bar-2.3.4.yaml"
schema = load_schema(uri)
assert schema["id"] == uri
```

9.3 Installing resources via entry points

The asdf package also offers an entry point for installing resource mapping plugins. This installs a package's resources automatically without requiring calls to the AsdfConfig method. The entry point is called `asdf.resource_mappings` and expects to receive a method that returns a list of `Mapping` instances.

For example, let's say we're creating a package named `asdf-foo-schemas` that provides the same schemas described in the previous section. Our directory structure might look something like this:

```
asdf-foo-schemas
├─ pyproject.toml
└─ src
   └─ asdf_foo_schemas
      ├─ __init__.py
      ├─ integration.py
      └─ schemas
         ├─ __init__.py
         ├─ foo-1.0.0.yaml
         ├─ README
         └─ nested
            ├─ __init__.py
            ├─ bar-2.3.4.yaml
            └─ baz-8.1.1.yaml
```

`pyproject.toml` is the preferred central configuration file for Python build and development systems. However, it is also possible to write configuration to a `setup.cfg` file (used by `setuptools`) placed in the root directory of the project. This documentation will cover both options.

In `integration.py`, we'll define the entry point method and have it return a list with a single element, our `DirectoryResourceMapping` instance:

```
# integration.py
from pathlib import Path

from asdf.resource import DirectoryResourceMapping


def get_resource_mappings():
    # Get path to schemas directory relative to this file
    schemas_path = Path(__file__).parent / "schemas"
    mapping = DirectoryResourceMapping(
        schemas_path,
        "asdf://example.com/example-project/schemas/",
        recursive=True,
        filename_pattern="*.yaml",
        stem_filename=True
```

(continues on next page)

(continued from previous page)

```
    )
    return [mapping]
```

Then in `pyproject.toml`, define an `[project.entry-points]` section (or `[options.entry_points]` in `setup.cfg`) that identifies the method as an `asdf.resource_mappings` entry point:

```
[project.entry-points]
'asdf.resource_mappings' = { asdf_foo_schemas = 'asdf_foo_schemas.integration:get_resource_
↪mappings' }
```

```
[options.entry-points]
asdf.resource_mappings =
    asdf_foo_schemas = asdf_foo_schemas.integration:get_resource_mappings
```

After installing the package, it should be possible to load one of our schemas in a new session without any additional setup:

```
from asdf.schema import load_schema

uri = "asdf://example.com/example-project/schemas/nested/bar-2.3.4.yaml"
schema = load_schema(uri)
assert schema["id"] == uri
```

Note that the package will need to be configured to include the YAML files. There are multiple ways to accomplish this, but one easy option is to add `[tool.setuptools.package-data]` and `[tool.setuptools.package-dir]` sections to `pyproject.toml` (or `[options.package_data]` in `setup.cfg`) requesting that all files with a `.yaml` extension be installed:

```
[tool.setuptools]
packages = ["asdf_foo_schemas", "asdf_foo_schemas.resources"]

[tool.setuptools.package-data]
"asdf_foo_schemas.resources" = ["resources/**/*.yaml"]

[tool.setuptools.package-dir]
"" = "src"
"asdf_foo_schemas.resources" = "resources"
```

```
[options.package_data]
* = *.yaml
```

9.3.1 Entry point performance considerations

For the good of `asdf` users everywhere, it's important that entry point methods load as quickly as possible. All resource URIs must be loaded before reading an ASDF file, so any entry point method that lingers will introduce a delay to the initial call to `asdf.open`. For that reason, we recommend to minimize the number of imports that occur in the module containing the entry point method, particularly imports of modules outside of the Python standard library or `asdf` itself. When resources are stored in a filesystem, it's also helpful to delay reading a file until its URI is actually requested, which may not occur in a given session. The DirectoryResourceMapping class is implemented with this behavior.

CHAPTER

TEN

CONVERTERS

The `Converter` interface defines a mapping between tagged objects in the ASDF tree and their corresponding Python object(s). Typically a Converter will map one YAML tag to one Python type, but the interface also supports many-to-one and many-to-many mappings. A Converter provides the software support for a tag and is responsible for both converting from parsed YAML to more complex Python objects and vice versa.

10.1 The Converter interface

Every Converter implementation must provide two required properties and two required methods:

`Converter.tags` - a list of tag URIs or URI patterns handled by the converter. Patterns may include the wildcard character `*`, which matches any sequence of characters up to a `/`, or `**`, which matches any sequence of characters. The `uri_match` method can be used to test URI patterns.

`Converter.types` - a list of Python types or fully-qualified Python type names handled by the converter. Note that a string name must reflect the actual location of the class's implementation and not just a module where it is imported for convenience. For example, if class `Foo` is implemented in `example_package.foo.Foo` but imported as `example_package.Foo` for convenience, it is the former name that must be used. The `get_class_name` method will return the name that `asdf` expects.

The string type name is recommended over a type object for performance reasons, see *Entry point performance considerations*.

`Converter.to_yaml_tree` - a method that accepts a complex Python object and returns a simple node object (typically a `dict`) suitable for serialization to YAML. The node is permitted to contain nested complex objects; these will in turn be passed to other `to_yaml_tree` methods in other Converters.

`Converter.from_yaml_tree` - a method that accepts a simple node object from parsed YAML and returns the appropriate complex Python object. Nested nodes in the received node will have already been converted to complex objects by other calls to `from_yaml_tree` methods, except where reference cycles are present – see *Reference cycles* for information on how to handle that situation.

Additionally, the Converter interface includes a method that must be implemented when some logic is required to select the tag to assign to a `to_yaml_tree` result:

`Converter.select_tag` - a method that accepts a complex Python object and a list candidate tags and returns the tag that should be used to serialize the object.

10.2 A simple example

Say we have a Python class, `Rectangle`, that we wish to serialize to an ASDF file. A `Rectangle` instance has two attributes, width and height, and a convenient method that computes its area:

```
# in module example_package.shapes
class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def get_area(self):
        return self.width * self.height
```

We'll need to designate a tag URI to represent this object's type in the ASDF tree – let's use `asdf://example.com/example-project/tags/rectangle-1.0.0`. Here is a simple Converter implementation for this type and tag:

```
from asdf.extension import Converter

class RectangleConverter(Converter):
    tags = ["asdf://example.com/shapes/tags/rectangle-1.0.0"]
    types = ["example_package.shapes.Rectangle"]

    def to_yaml_tree(self, obj, tag, ctx):
        return {
            "width": obj.width,
            "height": obj.height,
        }

    def from_yaml_tree(self, node, tag, ctx):
        from example_package.shapes import Rectangle

        return Rectangle(node["width"], node["height"])
```

Note that import of the `Rectangle` class has been deferred to inside the `from_yaml_tree` method. This is a performance consideration that is discussed in *Entry point performance considerations*.

In order to use this Converter, we'll need to create a simple extension around it and install that extension:

```
import asdf
from asdf.extension import Extension

class ShapesExtension(Extension):
    extension_uri = "asdf://example.com/shapes/extensions/shapes-1.0.0"
    converters = [RectangleConverter()]
    tags = ["asdf://example.com/shapes/tags/rectangle-1.0.0"]

asdf.get_config().add_extension(ShapesExtension())
```

Now we can include a Rectangle object in an `AsdfFile` tree and write out a file:

```
with asdf.AsdfFile() as af:
    af["rect"] = Rectangle(5, 4)
    af.write_to("test.asdf")
```

The portion of the ASDF file that represents the rectangle looks like this:

```
rect: !<asdf://example.com/shapes/tags/rectangle-1.0.0> {height: 4, width: 5}
```

10.3 Multiple tags

Now say we want to map our one Rectangle class to one of two tags, either rectangle-1.0.0 or square-1.0.0. We'll need to add square-1.0.0 to the converter's list of tags and implement a `select_tag` method:

```
RETANGLE_TAG = "asdf://example.com/shapes/tags/rectangle-1.0.0"
SQUARE_TAG = "asdf://example.com/shapes/tags/square-1.0.0"

class RectangleConverter(Converter):
    tags = [RECTANGLE_TAG, SQUARE_TAG]
    types = ["example_package.shapes.Rectangle"]

    def select_tag(self, obj, tags, ctx):
        if obj.width == obj.height:
            return SQUARE_TAG
        else:
            return RECTANGLE_TAG

    def to_yaml_tree(self, obj, tag, ctx):
        if tag == SQUARE_TAG:
            return {
                "side_length": obj.width,
            }
        else:
            return {
                "width": obj.width,
                "height": obj.height,
            }

    def from_yaml_tree(self, node, tag, ctx):
        from example_package.shapes import Rectangle

        if tag == SQUARE_TAG:
            return Rectangle(node["side_length"], node["side_length"])
        else:
            return Rectangle(node["width"], node["height"])
```

10.4 Reference cycles

Special considerations must be made when deserializing a tagged object that contains a reference to itself among its descendants. Consider a `fractions.Fraction` subclass that maintains a reference to its multiplicative inverse:

```
# in the example_project.fractions module
class FractionWithInverse(fractions.Fraction):
    def __init__(self, *args, **kwargs):
```

(continues on next page)

(continued from previous page)

```
        self._inverse = None

    @property
    def inverse(self):
        return self._inverse

    @inverse.setter
    def inverse(self, value):
        self._inverse = value
```

The inverse of the inverse of a fraction is the fraction itself, we might wish to construct the objects in the following way:

```
f1 = FractionWithInverse(3, 5)
f2 = FractionWithInverse(5, 3)
f1.inverse = f2
f2.inverse = f1
```

Which creates an "infinite loop" between the two fractions. An ordinary Converter wouldn't be able to deserialize this, since each fraction requires that the other be deserialized first! Let's see what happens when we define our `from_yaml_tree` method in a naive way:

```
class FractionWithInverseConverter(Converter):
    tags = ["asdf://example.com/fractions/tags/fraction-1.0.0"]
    types = ["example_project.fractions.FractionWithInverse"]

    def to_yaml_tree(self, obj, tag, ctx):
        return {
            "numerator": obj.width,
            "denominator": obj.height,
            "inverse": obj.inverse,
        }

    def from_yaml_tree(self, node, tag, ctx):
        from example_project.fractions import FractionWithInverse

        obj = FractionWithInverse(
            tree["numerator"],
            tree["denominator"]
        )
        obj.inverse = tree["inverse"]
        return obj
```

After adding this Converter to an Extension and installing it, the fraction will serialize correctly:

```
with asdf.AsdfFile({"fraction": f1}) as af:
    af.write_to("with_inverse.asdf")
```

But upon deserialization, we notice a problem:

```
with asdf.open("with_inverse.asdf") as af:
    reconstituted_f1 = af["fraction"]
```

(continues on next page)

(continued from previous page)

```
assert reconstituted_f1.inverse.inverse is asdf.treeutil.PendingValue
```

The presence of `PendingValue` is asdf's way of telling us that the value corresponding to the key `inverse` was not fully deserialized at the time that we retrieved it. We can handle this situation by making our `from_yaml_tree` a generator function:

```
def from_yaml_tree(self, node, tag, ctx):
    from example_project.fractions import FractionWithInverse

    obj = FractionWithInverse(
        tree["numerator"],
        tree["denominator"]
    )
    yield obj
    obj.inverse = tree["inverse"]
```

The generator version of `from_yaml_tree` yields the partially constructed `FractionWithInverse` object before setting its inverse property. This allows `asdf` to proceed to constructing the inverse `FractionWithInverse` object, and resume the original `from_yaml_tree` execution only when the inverse is actually available.

With this modification we can successfully deserialize our ASDF file:

```
with asdf.open("with_inverse.asdf") as af:
        reconstituted_f1 = ff["fraction"]

assert reconstituted_f1.inverse.inverse is reconstituted_f1
```

10.5 Entry point performance considerations

For the good of `asdf` users everywhere, it's important that entry point methods load as quickly as possible. All extensions must be loaded before reading an ASDF file, and therefore all converters are created as well. Any converter module or `__init__` method that lingers will introduce a delay to the initial call to `asdf.open`. For that reason, we recommend that converter authors minimize the number of imports that occur in the module containing the Converter implementation, and defer imports of serializable types to within the `from_yaml_tree` method. This will prevent the type from ever being imported when reading ASDF files that do not contain the associated tag.

CHAPTER

ELEVEN

EXTENSIONS

An ASDF "extension" is a supplement to the core ASDF specification that describes additional YAML tags or binary block compressors which may be used when writing files. In this library, extensions implement the `Extension` interface and can be installed manually by the user or automatically by a package using Python's entry points mechanism.

11.1 Extension features

11.1.1 Basics

Every extension to ASDF must be uniquely identified by a URI; this URI is written to the file's metadata when the extension is used and allows software to determine if the necessary extensions are installed when the file is read. An ASDF extension implementation intended for use with this library must, at a minimum, implement the `Extension` interface and provide its URI as a property:

```
from asdf.extension import Extension

class FooExtension(Extension):
    extension_uri = "asdf://example.com/example-project/extensions/foo-1.0.0"
```

Note that this is an "empty" extension that does not extend the library in any meaningful way; other attributes must be implemented to actually support additional tags and/or compressors. Read on for a description of the rest of the Extension interface.

11.1.2 Additional tags

In order to implement support for additional YAML tags, an Extension subclass must provide both a list of relevant tags and a list of `Converter` instances that translate objects with those tags to and from YAML. These lists are provided in the `tags` and `converters` properties, respectively:

```
from asdf.extension import Extension, Converter

class FooConverter(Converter):
    # ...

class FooExtension(Extension):
    extension_uri = "asdf://example.com/example-project/extensions/foo-1.0.0"
    tags = ["asdf://example.com/example-project/tags/foo-1.0.0"]
    converters = [FooConverter()]
```

The implementation of a Converter is a topic unto itself and is discussed in detail in *Converters*.

The Extension implemented above will happily convert between `foo-1.0.0` tagged YAML objects and the appropriate Python representation, but it will not perform any schema validation. In order to associate the tag with a schema, we'll need to provide a `TagDefinition` object instead of just a string:

```
from asdf.extension import Extension, Converter, TagDefinition

class FooConverter(Converter):
    # ...

class FooExtension(Extension):
    extension_uri = "asdf://example.com/example-project/extensions/foo-1.0.0"
    tags = [
        TagDefinition(
            "asdf://example.com/example-project/tags/foo-1.0.0",
            schema_uri="asdf://example.com/example-project/schemas/foo-1.0.0",
        )
    ]
    converters = [FooConverter()]
```

11.1.3 Additional block compressors

Binary block compressors implement the `Compressor` interface and are included in an extension via the `compressors` property:

```
from asdf.extension import Extension, Compressor

class FooCompressor(Compressor):
    # ...

class FooExtension(Extension):
    extension_uri = "asdf://example.com/example-project/extensions/foo-1.0.0"
    compressors = [FooCompressor()]
```

See *Binary block compressors* for details on implementing the Compressor interface.

11.1.4 Additional YAML tag handles

The YAML format permits use of "tag handles" as shorthand prefixes in tags. For example, these two YAML files are equivalent:

```
%YAML 1.1
---
value: !<asdf://example.com/example-project/tags/foo-1.0.0>
  # etc
...
```

```
%YAML 1.1
%TAG !example! asdf://example.com/example-project/tags/
---
value: !example!foo-1.0.0
```

(continues on next page)

(continued from previous page)

```
  # etc
...
```

In both cases the `value` object has tag asdf://example.com/example-project/tags/foo-1.0.0, but in the second example the tag is abbreviated as `!example!foo-1.0.0` through use of a handle. This has no impact on the interpretation of the file but can make the raw ASDF tree easier to read for humans.

Tag handles can be defined in the `yaml_tag_handles` property of an extension:

```
from asdf.extension import Extension

class FooExtension(Extension):
    extension_uri = "asdf://example.com/example-project/extensions/foo-1.0.0"
    yaml_tag_handles = {
        "!example!": "asdf://example.com/example-project/tags/"
    }
```

11.1.5 ASDF Standard version requirement

Some extensions may only work with specific version(s) of the ASDF Standard – for example, the schema associated with one of an extension's tags may reference specific versions of ASDF core tags. This requirement can be expressed as a PEP 440 version specifier in an Extension's `asdf_standard_requirement` property:

```
from asdf.extension import Extension

class FooExtension(Extension):
    extension_uri = "asdf://example.com/example-project/extensions/foo-1.0.0"
    asdf_standard_requirement = ">= 1.2.0, < 1.5.0"
```

Now the extension will only be used with ASDF Standard 1.3.0 and 1.4.0 files.

11.1.6 Legacy class names

Previous versions of this library referred to extensions by their Python class names instead of by URI. These class names were written to ASDF file metadata and allowed the library to warn users when an extension used to write the file was not available on read. Now the extension URI is written to the metadata, but to prevent warnings when reading older files, extension authors can provide an additional list of class names that previously identified the extension:

```
from asdf.extension import Extension

class FooExtension(Extension):
    extension_uri = "asdf://example.com/example-project/extensions/foo-1.0.0"
    legacy_class_names = [
        "foo_package.extensions.FooExtension",
    ]
```

11.1.7 Making converted object's contents visible to `info` and `search`

If the object produced by the extension supports a class method `__asdf_traverse__` then it can be used by those tools to expose the contents of the object. That method should accept no arguments and return either a dict of attributes and their values, or a list if the object itself is list-like.

11.2 Installing an extension

Once an extension is implemented, it must be installed so that the `asdf` library knows to use it. There are two options for installing an extension: manually per session using `AsdfConfig`, or automatically for every session using the `asdf.extensions` entry point

11.2.1 Installing extensions via AsdfConfig

The simplest way to install an extension is to add it at runtime using the `AsdfConfig.add_extension` method. For example, the following code defines and installs a minimal extension:

```
import asdf
from asdf.extension import Extension

class FooExtension(Extension):
    extension_uri = "asdf://example.com/example-project/extensions/foo-1.0.0"

asdf.get_config().add_extension(FooExtension())
```

Now the extension will be available when working with ASDF files, but only for the duration of the current Python session.

11.2.2 Installing extensions via entry points

The `asdf` package also offers an entry point for installing extensions This registers a package's extensions automatically on package install without requiring calls to the AsdfConfig method. The entry point is called `asdf.extensions` and expects to receive a method that returns a list of `Extension` instances.

For example, let's say we're creating a package named `asdf-foo-extension` that provides the not-particularly-useful `FooExtension` from the previous section. We'll need to define an entry point method that returns a list containing an instance of `FooExtension`:

```
def get_extensions():
    return [FooExtension()]
```

We'll assume that method is located in the module `asdf_foo_extension.integration`.

Next, in the package's `pyproject.toml`, define a `[project.entry-points]` section (or `[options.entry_points]` in `setup.cfg`) that identifies the method as an `asdf.extensions` entry point:

```
[project.entry-points]
'asdf.extensions' = { asdf_foo_extension = 'asdf_foo_extension.integration:get_extensions' }
```

```
[options.entry_points]
asdf.extensions =
    asdf_foo_extension = asdf_foo_extension.integration:get_extensions
```

After installing the package, the extension should be automatically available in any new Python session.

11.2.3 Entry point performance considerations

For the good of `asdf` users everywhere, it's important that entry point methods load as quickly as possible. All extensions must be loaded before reading an ASDF file, so any entry point method that lingers will introduce a delay to the initial call to `asdf.open`. For that reason, we recommend that extension authors minimize the number of imports that occur in the module containing the entry point method, particularly imports of modules outside of the Python standard library or `asdf` itself.

11.3 Populating an extension from a manifest

An "extension manifest" is a language-independent description of an ASDF extension (little 'e') that includes information such as the extension URI, list of tags, ASDF Standard requirement, etc. Instructions on writing a manifest can be found in *Extension manifests*, but once written, we'll still need a Python Extension (big 'E') whose content mirrors the manifest. Rather than duplicate that information in Python code, we recommend use of the `ManifestExtension` class, which reads a manifest and maps its content to the appropriate Extension interface properties.

Assuming the manifest is installed as a resource (see *Resources and resource mappings*), an extension instance can be created using the `from_uri` factory method:

```
from asdf.extension import ManifestExtension

extension = ManifestExtension.from_uri("asdf://example.com/example-project/manifests/foo-1.0.
↪0")
```

Compressors and converters can be included in the extension by adding them as keyword arguments:

```
from asdf.extension import ManifestExtension

extension = ManifestExtension.from_uri(
    "asdf://example.com/example-project/manifests/foo-1.0.0",
    converters=[FooConverter()],
    compressors=[FooCompressor()],
)
```

The extension may then be installed by one of the two methods described above.

11.3.1 Warning on ManifestExtension and entry points

When implementing a package that automatically installs a ManifestExtension, we'll need to utilize both the `asdf.resource_mappings` entry point (to install the manifest) and the `asdf.extensions` entry point (to install the extension). Because the manifest must be installed before the extension can be instantiated, it's easy to end up trapped in an import loop. For example, this seemingly innocuous set of entry point methods cannot be successfully loaded:

```
from asdf.extension import ManifestExtension

RESOURCES = {
    "asdf://example.com/example-project/manifests/foo-1.0.0": open("foo-1.0.0.yaml").read()
}
```

(continues on next page)

(continued from previous page)

```
def get_resource_mappings():
    return [RESOURCES]

EXTENSION = ManifestExtension.from_uri("asdf://example.com/example-project/manifests/foo-1.0.
↪0")

def get_extensions():
    return [EXTENSION]
```

When the module is imported, `ManifestExtension.from_uri` asks the `asdf` library to load all available resources so that it can retrieve the manifest content. But loading the resources requires importing this module to get at the `get_resource_mappings` method, so now we're stuck!

The solution is to instantiate the ManifestExtension inside of its entry point method:

```
def get_extensions():
    return [
        ManifestExtension.from_uri("asdf://example.com/example-project/manifests/foo-1.0.0")
    ]
```

This is not as inefficient as it might seem, since the `asdf` library only calls the method once and reuses a cached result thereafter.

CHAPTER

TWELVE

EXTENSION MANIFESTS

An extension "manifest" is a YAML document that defines an extension in a language-independent way. Use of a manifest is recommended for ASDF extensions that are intended to be implemented by ASDF libraries in multiple languages, so that other implementers do not need to go spelunking through Python code to discover the tags and schemas that are included in the extension. This library provides support for automatically populating a `Extension` object from a manifest; see *Populating an extension from a manifest* for more information.

12.1 Anatomy of a manifest

Here is an example of a simple manifest that describes an extension with one tag and schema:

```
%YAML 1.1
---
id: asdf://example.com/example-project/manifests/example-1.0.0
extension_uri: asdf://example.com/example-project/extensions/example-1.0.0
title: Example extension 1.0.0
description: Tags for example objects.
asdf_standard_requirement:
    gte: 1.3.0
    lt: 1.5.0
tags:
  - tag_uri: asdf://example.com/example-project/tags/foo-1.0.0
    schema_uri: asdf://example.com/example-project/schemas/foo-1.0.0
...
```

```
id: asdf://example.com/example-project/manifests/example-1.0.0
```

The `id` property contains the URI that uniquely identifies our manifest. This URI is how we'll refer to the manifest document's content when using the `asdf` library.

```
extension_uri: asdf://example.com/example-project/extensions/example-1.0.0
```

The `extension_uri` property contains the URI of the extension that the manifest describes. This is the URI written to ASDF file metadata to document that an extension was used when writing the file.

```
title: Example extension 1.0.0
description: Tags for example objects.
```

`title` and `description` are optional documentation properties.

```
asdf_standard_requirement:
    gte: 1.3.0
    lt: 1.5.0
```

The optional `asdf_standard_requirement` property describes the ASDF Standard versions that are compatible with this extension. The `gte` and `lt` properties are used here to restrict ASDF Standard versions to greater-than-or-equal 1.3.0 and less-than 1.5.0, respectively. `gt` and `lte` properties are also available.

```
tags:
  - tag_uri: asdf://example.com/example-project/tags/foo-1.0.0
    schema_uri: asdf://example.com/example-project/schemas/foo-1.0.0
```

The `tags` property contains a list of objects, each representing a new tag that the extension brings to ASDF. The `tag_uri` property contains the tag itself, while the (optional, but recommended) `schema_uri` property contains the URI of a schema that can be used to validate objects with that tag. Tag objects may also include `title` and `description` documentation properties.

12.2 Validating a manifest

This library includes a schema, `asdf://asdf-format.org/core/schemas/extension_manifest-1.0.0`, that can be used to validate a manifest document:

```
import asdf
import yaml

schema = asdf.schema.load_schema("asdf://asdf-format.org/core/schemas/extension_manifest-1.0.
↪0")
manifest = yaml.safe_load(open("path/to/manifests/example-1.0.0.yaml").read())
asdf.schema.validate(manifest, schema=schema)
```

CHAPTER

THIRTEEN

BINARY BLOCK COMPRESSORS

The `Compressor` interface provides an implementation of a compression algorithm that can be used to transform binary blocks in an `AsdfFile`. Each Compressor must provide a 4-byte compression code that identifies the algorithm. Once the Compressor is installed as part of an Extension plugin, this code will be available to users as an argument to `set_array_compression` and the `all_array_compression` argument to `write_to` and `update`.

See *Additional block compressors* for details on including a Compressor in an extension.

13.1 The Compressor interface

Every Compressor implementation must provide one required property and two required methods:

`Compressor.label` - A 4-byte compression code. This code is used by users to select a compression algorithm and also stored in the binary block header to identify the algorithm that was applied to the block's data.

`Compressor.compress` - The method that transforms the block's bytes before they are written to an ASDF file. The positional argument is a `memoryview` object which is guaranteed to be 1D and contiguous. Compressors must be prepared to handle `memoryview.itemsize` > 1. Any keyword arguments are passed through from the user and may be used to tune the compression algorithm. `compress` methods have no return value and instead are expected to yield bytes-like values until the input data has been fully compressed.

`Compressor.decompress` - The method that transforms the block's bytes after they are read from an ASDF file. The first positional argument is an `Iterable` of bytes-like objects that each contain a chunk of the compressed input data. The second positional argument is a pre-allocated output array where the decompressed bytes should be written. The method is expected to return the number of bytes written to the output array.

13.2 Entry point performance considerations

For the good of `asdf` users everywhere, it's important that entry point methods load as quickly as possible. All extensions must be loaded before reading an ASDF file, and therefore all compressors are created as well. Any compressor module or `__init__` method that lingers will introduce a delay to the initial call to `asdf.open`. For that reason, we recommend that compressor authors minimize the number of imports that occur in the module containing the Compressor implementation, and defer imports of compression libraries to inside the `Comrpessor.compress` and `Compressor.decompress` methods. This will prevent the library from ever being imported when reading ASDF files that do not utilize the Compressor's algorithm.

CHAPTER

FOURTEEN

DEPRECATED EXTENSION API

This page documents the original `asdf` extension API, which has been deprecated in favor of *Extensions*. Since support for the deprecated API will be removed in `asdf` 3.0, we recommend that all new extensions be implemented with the new API.

Extensions provide a way for ASDF to represent complex types that are not defined by the ASDF standard. Examples of types that require custom extensions include types from third-party libraries, user-defined types, and complex types that are part of the Python standard library but are not handled in the ASDF standard. From ASDF's perspective, these are all considered 'custom' types.

Supporting new types in ASDF is easy. Three components are required:

1. A YAML Schema file for each new type.
2. A tag class (inheriting from `asdf.CustomType`) corresponding to each new custom type. The class must override `to_tree` and `from_tree` from `asdf.CustomType` in order to define how ASDF serializes and deserializes the custom type.
3. A Python class to define an "extension" to ASDF, which is a set of related types. This class must implement the `asdf.AsdfExtension` abstract base class. In general, a third-party library that defines multiple custom types can group them all in the same extension.

Note: The mechanisms of tag classes and extension classes are specific to this particular implementation of ASDF. As of this writing, this is the only complete implementation of the ASDF Standard. However, other language implementations may use other mechanisms for processing custom types.

All implementations of ASDF, regardless of language, will make use of the same schemas for abstract data type definitions. This allows all ASDF files to be language-agnostic, and also enables interoperability.

14.1 An Example

As an example, we will write an extension for ASDF that allows us to represent Python's standard `fractions.Fraction` class for representing rational numbers. We will call our new ASDF type `fraction`.

First, the YAML Schema, defining the type as a pair of integers:

```
%YAML 1.1
---
$schema: "http://stsci.edu/schemas/yaml-schema/draft-01"
id: "http://nowhere.org/schemas/custom/fraction-1.0.0"
title: An example custom type for handling fractions

```

(continues on next page)

(continued from previous page)

```
tag: "tag:nowhere.org:custom/fraction-1.0.0"
type: array
items:
  type: integer
minItems: 2
maxItems: 2
...
```

Then, the Python implementation of the tag class and extension class. See the `asdf.CustomType` and `asdf.AsdfExtension` documentation for more information:

```
import os

import asdf
from asdf import util

import fractions

class FractionType(asdf.CustomType):
    name = 'fraction'
    organization = 'nowhere.org'
    version = (1, 0, 0)
    standard = 'custom'
    types = [fractions.Fraction]

    @classmethod
    def to_tree(cls, node, ctx):
        return [node.numerator, node.denominator]

    @classmethod
    def from_tree(cls, tree, ctx):
        return fractions.Fraction(tree[0], tree[1])

class FractionExtension(asdf.AsdfExtension):
    @property
    def types(self):
        return [FractionType]

    @property
    def tag_mapping(self):
        return [('tag:nowhere.org:custom',
                 'http://nowhere.org/schemas/custom{tag_suffix}')]

    @property
    def url_mapping(self):
        return [('http://nowhere.org/schemas/custom/',
                 util.filepath_to_url(os.path.dirname(__file__))
                 + '/{url_suffix}.yaml')]
```

Note that the method `to_tree` of the tag class `FractionType` defines how the library converts `fractions.Fraction` into a tree that can be stored by ASDF. Conversely, the method `from_tree` defines how the library reads a serialized representation of the object and converts it back into an instance of `fractions.Fraction`.

Note that the values of the `name`, `organization`, `standard`, and `version` fields are all reflected in the `id` and `tag` definitions in the schema.

Note also that the base of the `tag` value (up to the `name` and `version` components) is reflected in `tag_mapping` property of the `FractionExtension` type, which is used to map tags to URLs. The `url_mapping` is used to map URLs (of the same form as the `id` field in the schema) to the actual location of a schema file.

Once these classes and the schema have been defined, we can save an ASDF file using them:

```
tree = {'fraction': fractions.Fraction(10, 3)}

with asdf.AsdfFile(tree, extensions=FractionExtension()) as ff:
    ff.write_to("test.asdf")
```

test.asdf

```
#ASDF 1.0.0
#ASDF_STANDARD 1.5.0
%YAML 1.1
%TAG ! tag:stsci.edu:asdf/
--- !core/asdf-1.1.0
asdf_library: !core/software-1.0.0 {author: The ASDF Developers, homepage: 'http://github.
↪com/asdf-format/asdf',
  name: asdf, version: 2.12.1.dev90+g784740b}
history:
  extensions:
  - !core/extension_metadata-1.0.0
    extension_class: asdf.extension.BuiltinExtension
    software: !core/software-1.0.0 {name: asdf, version: 2.12.1.dev90+g784740b}
  - !core/extension_metadata-1.0.0 {extension_class: builtins.FractionExtension}
fraction: !<tag:nowhere.org:custom/fraction-1.0.0> [10, 3]
...
```

14.2 Defining custom types

In the example above, we showed how to create an extension that is capable of serializing `fractions.Fraction`. The custom tag type that we created was defined as a subclass of `asdf.CustomType`.

14.2.1 Custom type attributes

We overrode the following attributes of `CustomType` in order to define `FractionType` (each bullet is also a link to the API documentation):

- `name`
- `organization`
- `version`
- `standard`
- `types`

Each of these attributes is important, and each is described in more detail in the linked API documentation.

The choice of `name` should be descriptive of the custom type that is being serialized. The choice of `organization`, and `standard` is fairly arbitrary, but also important. Custom types that are provided by the same package should be grouped into the same `standard` and `organization`.

These three values, along with the `version`, are used to define the YAML tag that will mark the serialized type in ASDF files. In our example, the tag becomes `tag:nowhere.org:custom/fraction-1.0.0`. The tag is important when defining the `asdf.AsdfExtension` subclass.

Critically, these values must all be reflected in the associated schema.

14.2.2 Custom type methods

In addition to the attributes mentioned above, we also overrode the following methods of `CustomType` (each bullet is also a link to the API documentation):

- `to_tree`
- `from_tree`

The `to_tree` method defines how an instance of a custom data type is converted into data structures that represent a YAML tree that can be serialized to a file.

The `from_tree` method defines how a YAML tree can be converted back into an instance of the original custom data type.

In the example above, we used a `list` to contain the important attributes of `fractions.Fraction`. However, this choice is fairly arbitrary, as long as it is consistent between the way that `to_tree` and `from_tree` are defined. For example, we could have also chosen to use a `dict`:

```
import asdf
import fractions

class FractionType(asdf.CustomType):
    name = 'fraction'
    organization = 'nowhere.org'
    version = (1, 0, 0)
    standard = 'custom'
    types = [fractions.Fraction]

    @classmethod
    def to_tree(cls, node, ctx):
        return dict(numerator=node.numerator,
                    denominator=node.denominator)

    @classmethod
    def from_tree(cls, tree, ctx):
        return fractions.Fraction(tree['numerator'],
                                  tree['denominator'])
```

In this case, the associated schema would look like the following:

```
%YAML 1.1
---
$schema: "http://stsci.edu/schemas/yaml-schema/draft-01"
id: "http://nowhere.org/schemas/custom/fraction-1.0.0"
```

(continues on next page)

(continued from previous page)

```
title: An example custom type for handling fractions

tag: "tag:nowhere.org:custom/fraction-1.0.0"
type: object
properties:
  numerator:
    type: integer
  denominator:
    type: integer
...
```

We can compare the output using this representation to the example above:

test.asdf

```
#ASDF 1.0.0
#ASDF_STANDARD 1.5.0
%YAML 1.1
%TAG ! tag:stsci.edu:asdf/
--- !core/asdf-1.1.0
asdf_library: !core/software-1.0.0 {author: The ASDF Developers, homepage: 'http://github.
↪com/asdf-format/asdf',
  name: asdf, version: 2.12.1.dev90+g784740b}
history:
  extensions:
  - !core/extension_metadata-1.0.0
    extension_class: asdf.extension.BuiltinExtension
    software: !core/software-1.0.0 {name: asdf, version: 2.12.1.dev90+g784740b}
  - !core/extension_metadata-1.0.0 {extension_class: builtins.FractionExtension}
fraction: !<tag:nowhere.org:custom/fraction-1.0.0> {denominator: 3, numerator: 10}
...
```

14.2.3 Serializing more complex types

Sometimes the custom types that we wish to represent in ASDF themselves have attributes which are also custom types. As a somewhat contrived example, consider a 2D cartesian coordinate that uses `fraction.Fraction` to represent each of the components. We will call this type `Fractional2DCoordinate`.

First we need to define a schema to represent this new type:

```
%YAML 1.1
---
$schema: "http://stsci.edu/schemas/yaml-schema/draft-01"
id: "http://nowhere.org/schemas/custom/fractional_2d_coord-1.0.0"
title: An example custom type for handling components

tag: "tag:nowhere.org:custom/fractional_2d_coord-1.0.0"
type: object
properties:
  x:
```

(continues on next page)

(continued from previous page)

```
    $ref: fraction-1.0.0
  y:
    $ref: fraction-1.0.0
...
```

Note that in the schema, the `x` and `y` attributes are expressed as references to our `fraction-1.0.0` schema. Since both of these schemas are defined under the same standard and organization, we can simply use the name and version of the `fraction-1.0.0` schema to refer to it. However, if the reference type was defined in a different organization and standard, it would be necessary to use the entire YAML tag in the reference (e.g. `tag:nowhere.org:custom/fraction-1.0.0`). Relative tag references are also allowed where appropriate.

We also need to define the custom tag type that corresponds to our new type:

```
import asdf

class Fractional2DCoordinateType(asdf.CustomType):
    name = 'fractional_2d_coord'
    organization = 'nowhere.org'
    version = (1, 0, 0)
    standard = 'custom'
    types = [Fractional2DCoordinate]

    @classmethod
    def to_tree(cls, node, ctx):
        tree = dict()
        tree['x'] = node.x
        tree['y'] = node.y
        return tree

    @classmethod
    def from_tree(cls, tree, ctx):
        coord = Fractional2DCoordinate()
        coord.x = tree['x']
        coord.y = tree['y']
        return coord
```

In previous versions of this library, it was necessary for our `Fractional2DCoordinateType` class to call `yamlutil` functions explicitly to convert the `x` and `y` components to and from their tree representations. Now, the library will automatically convert nested custom types before calling `from_tree`, and after receiving the result from `to_tree`.

Since `Fractional2DCoordinateType` shares the same `organization` and `standard` as `FractionType`, it can be added to the same extension class:

```
class FractionExtension(asdf.AsdfExtension):
    @property
    def types(self):
        return [FractionType, Fractional2DCoordinateType]

    @property
    def tag_mapping(self):
        return [('tag:nowhere.org:custom',
                 'http://nowhere.org/schemas/custom{tag_suffix}')]
```

(continues on next page)

(continued from previous page)

```
    @property
    def url_mapping(self):
        return [('http://nowhere.org/schemas/custom/',
                 util.filepath_to_url(os.path.dirname(__file__))
                 + '/{url_suffix}.yaml')]
```

Now we can use this extension to create an ASDF file:

```
coord = Fractional2DCoordinate()
coord.x = fractions.Fraction(22, 7)
coord.y = fractions.Fraction(355, 113)

tree = {'coordinate': coord}

with asdf.AsdfFile(tree, extensions=FractionExtension()) as ff:
    ff.write_to("coord.asdf")
```

coord.asdf

```
#ASDF 1.0.0
#ASDF_STANDARD 1.5.0
%YAML 1.1
%TAG ! tag:stsci.edu:asdf/
--- !core/asdf-1.1.0
asdf_library: !core/software-1.0.0 {author: The ASDF Developers, homepage: 'http://github.
↪com/asdf-format/asdf',
  name: asdf, version: 2.12.1.dev90+g784740b}
history:
  extensions:
  - !core/extension_metadata-1.0.0
    extension_class: asdf.extension.BuiltinExtension
    software: !core/software-1.0.0 {name: asdf, version: 2.12.1.dev90+g784740b}
  - !core/extension_metadata-1.0.0 {extension_class: builtins.FractionExtension}
coordinate: !<tag:nowhere.org:custom/fractional_2d_coord-1.0.0>
  x: !<tag:nowhere.org:custom/fraction-1.0.0> {denominator: 7, numerator: 22}
  y: !<tag:nowhere.org:custom/fraction-1.0.0> {denominator: 113, numerator: 355}
...
```

Note that in the resulting ASDF file, the `x` and `y` components of our new `fraction_2d_coord` type are tagged as `fraction-1.0.0`.

14.2.4 Serializing reference cycles

Special considerations must be made when deserializing a custom type that contains a reference to itself among its descendants. Consider a `fractions.Fraction` subclass that maintains a reference to its multiplicative inverse:

```
class FractionWithInverse(fractions.Fraction):
    def __init__(self, *args, **kwargs):
        self._inverse = None

    @property
    def inverse(self):
        return self._inverse

    @inverse.setter
    def inverse(self, value):
        self._inverse = value
```

The inverse of the inverse of a fraction is the fraction itself, so you might wish to construct your objects in the following way:

```
f1 = FractionWithInverse(3, 5)
f2 = FractionWithInverse(5, 3)
f1.inverse = f2
f2.inverse = f1
```

Which creates an "infinite loop" between the two fractions. An ordinary `CustomType` wouldn't be able to deserialize this, since each object requires that the other be deserialized first! Let's see what happens when we define our `from_tree` method in a naive way:

```
class FractionWithInverseType(asdf.CustomType):
    name = 'fraction_with_inverse'
    organization = 'nowhere.org'
    version = (1, 0, 0)
    standard = 'custom'
    types = [FractionWithInverse]

    @classmethod
    def to_tree(cls, node, ctx):
        return {
            "numerator": node.numerator,
            "denominator": node.denominator,
            "inverse": node.inverse
        }

    @classmethod
    def from_tree(cls, tree, ctx):
        result = FractionWithInverse(
            tree["numerator"],
            tree["denominator"]
        )
        result.inverse = tree["inverse"]
        return result
```

After adding our type to the extension class, the tree will serialize correctly:

```
tree = {'fraction': f1}

with asdf.AsdfFile(tree, extensions=FractionExtension()) as ff:
    ff.write_to("with_inverse.asdf")
```

But upon deserialization, we notice a problem:

```
with asdf.open("with_inverse.asdf", extensions=FractionExtension()) as ff:
    reconstituted_f1 = ff["fraction"]

assert reconstituted_f1.inverse.inverse is asdf.treeutil.PendingValue
```

The presence of `PendingValue` is `asdf`'s way of telling you that the value corresponding to the key `inverse` was not fully deserialized at the time that you retrieved it. We can handle this situation by making our `from_tree` a generator function:

```
class FractionWithInverseType(asdf.CustomType):
    name = 'fraction_with_inverse'
    organization = 'nowhere.org'
    version = (1, 0, 0)
    standard = 'custom'
    types = [FractionWithInverse]

    @classmethod
    def to_tree(cls, node, ctx):
        return {
            "numerator": node.numerator,
            "denominator": node.denominator,
            "inverse": node.inverse
        }

    @classmethod
    def from_tree(cls, tree, ctx):
        result = FractionWithInverse(
            tree["numerator"],
            tree["denominator"]
        )
        yield result
        result.inverse = tree["inverse"]
```

The generator version of `from_tree` yields the partially constructed `FractionWithInverse` object before setting its inverse property. This allows asdf to proceed to constructing the inverse `FractionWithInverse` object, and resume the original `from_tree` execution only when the inverse is actually available.

With this new version of `from_tree`, we can successfully deserialize our ASDF file:

```
with asdf.open("with_inverse.asdf", extensions=FractionExtension()) as ff:
        reconstituted_f1 = ff["fraction"]

assert reconstituted_f1.inverse.inverse is reconstituted_f1
```

14.2.5 Assigning schema and tag versions

Authors of new tags and schemas should strive to use the conventions described by semantic versioning. Tags and schemas for types that have not been serialized before should begin at `1.0.0`. Versions for a particular tag type need not move in lock-step with other tag types in the same extension.

The patch version should be bumped for bug fixes and other minor, backwards-compatible changes. New features can be indicated with increments to the minor version, as long as they remain backwards compatible with older versions of the schema. Any changes that break backwards compatibility must be indicated by a major version update.

Since ASDF is intended to be an archival file format, authors of tags and schemas should work to ensure that ASDF files created with older extensions can continue to be processed. This means that every time a schema version is bumped (with the possible exception of patch updates), a **new** schema file should be created.

For example, if we currently have a schema for `xyz-1.0.0`, and we wish to make changes and bump the version to `xyz-1.1.0`, we should leave the original schema intact. A **new** schema file should be created for `xyz-1.1.0`, which can exist in parallel with the old file. The version of the corresponding tag type should be bumped to `1.1.0`.

For more details on the behavior of schema and tag versioning from a user perspective, see *Versioning and Compatibility*, and also *Custom types, extensions, and versioning*.

14.2.6 Explicit version support

To some extent schemas and tag classes will be closely tied to the custom data types that they represent. This means that in some cases API changes or other changes to the representation of the underlying types will force us to modify our schemas and tag classes. ASDF's schema versioning allows us to handle changes in schemas over time.

Let's consider an imaginary custom type called `Person` that we want to serialize in ASDF. The first version of `Person` was constructed using a first and last name:

```
person = Person('James', 'Webb')
print(person.first, person.last)
```

Our version 1.0.0 YAML schema for `Person` might look like the following:

```
%YAML 1.1
---
$schema: "http://stsci.edu/schemas/yaml-schema/draft-01"
id: "http://nowhere.org/schemas/custom/person-1.0.0"
title: An example custom type for representing a Person

tag: "tag:nowhere.org:custom/person-1.0.0"
type: array
items:
  type: string
minItems: 2
maxItems: 2
...
```

And our tag implementation would look something like this:

```
import asdf
from people import Person

class PersonType(asdf.CustomType):
```

(continues on next page)

(continued from previous page)

```
    name = 'person'
    organization = 'nowhere.org'
    version = (1, 0, 0)
    standard = 'custom'
    types = [Person]

    @classmethod
    def to_tree(cls, node, ctx):
        return [node.first, node.last]

    @classmethod
    def from_tree(cls, tree, ctx):
        return Person(tree[0], tree[1])
```

However, a newer version of `Person` now requires a middle name in the constructor as well:

```
person = Person('James', 'Edwin', 'Webb')
print(person.first, person.middle, person.last)
James Edwin Webb
```

So we update our YAML schema to version 1.1.0 in order to support newer versions of Person:

```
%YAML 1.1
---
$schema: "http://stsci.edu/schemas/yaml-schema/draft-01"
id: "http://nowhere.org/schemas/custom/person-1.1.0"
title: An example custom type for representing a Person

tag: "tag:nowhere.org:custom/person-1.1.0"
type: array
items:
  type: string
minItems: 3
maxItems: 3
...
```

We need to update our tag class implementation as well. However, we need to be careful. We still want to be able to read version 1.0.0 of our schema and be able to convert it to the newer version of `Person` objects. To accomplish this, we will make use of the `supported_versions` attribute for our tag class. This will allow us to declare explicit support for the schema versions our tag class implements.

Under the hood, `asdf` creates multiple copies of our `PersonType` tag class, each with a different `version` attribute corresponding to one of the supported versions. This means that in our new tag class implementation, we can condition our `from_tree` implementation on the value of `version` to determine which schema version should be used when reading:

```
import asdf
from people import Person

class PersonType(asdf.CustomType):
    name = 'person'
    organization = 'nowhere.org'
    version = (1, 1, 0)
```

(continues on next page)

(continued from previous page)

```
    supported_versions = [(1, 0, 0), (1, 1, 0)]
    standard = 'custom'
    types = [Person]

    @classmethod
    def to_tree(cls, node, ctx):
        return [node.first, node.middle, node.last]

    @classmethod
    def from_tree(cls, tree, ctx):
        # Handle the older version of the person schema
        if cls.version == (1, 0, 0):
            # Construct a Person object with an empty middle name field
            return Person(tree[0], '', tree[1])
        else:
            # The newer version of the schema stores the middle name too
            return person(tree[0], tree[1], tree[2])
```

Note that the implementation of `to_tree` is not conditioned on `cls.version` since we do not need to convert new `Person` objects back to the older version of the schema.

14.2.7 Handling subclasses

By default, if a custom type is serialized by an `asdf` tag class, then all subclasses of that type can also be serialized. However, no attributes that are specific to the subclass will be stored in the file. When reading the file, an instance of the base custom type will be returned instead of the subclass that was written.

To properly handle subclasses of custom types already recognized by `asdf`, it is necessary to implement a separate tag class that is specific to the subclass to be serialized.

Previous versions of this library implemented an experimental feature that allowed ADSF to serialize subclass attributes using the same tag class, but this feature was dropped as it produced files that were not portable.

14.3 Creating custom schemas

All custom types to be serialized by `asdf` require custom schemas. The best resource for creating ASDF schemas can be found in the ASDF Standard documentation.

In most cases, ASDF schemas will be included as part of a packaged software distribution. In these cases, it is important for the `url_mapping` of the corresponding `AsdfExtension` extension class to map the schema URL to an actual location on disk. However, it is possible for schemas to be hosted online as well, in which case the URL mapping can map (perhaps trivially) to an actual network location. See *Defining custom extension classes* for more information.

It is also important for packages that provide custom schemas to test them, both to make sure that they are valid, and to ensure that any examples they provide are also valid. See *Testing custom schemas* for more information.

14.4 Adding custom validators

A new type may also add new validation keywords to the schema language. This can be used to impose type-specific restrictions on the values in an ASDF file. This feature is used internally so a schema can specify the required datatype of an array.

To support custom validation keywords, set the `validators` member of a `CustomType` subclass to a dictionary where the keys are the validation keyword name and the values are validation functions. The validation functions are of the same form as the validation functions in the underlying `jsonschema` library, and are passed the following arguments:

- `validator`: A `jsonschema.Validator` instance.
- `value`: The value of the schema keyword.
- `instance`: The instance to validate. This will be made up of basic datatypes as represented in the YAML file (list, dict, number, strings), and not include any object types.
- `schema`: The entire schema that applies to instance. Useful to get other related schema keywords.

The validation function should either return `None` if the instance is valid or `yield` one or more `asdf.ValidationError` objects if the instance is invalid.

To continue the example from above, for the `FractionType` say we want to add a validation keyword "`simplified`" that, when `true`, asserts that the corresponding fraction is in simplified form:

```
from asdf import ValidationError

def validate_simplified(validator, simplified, instance, schema):
    if simplified:
        reduced = fraction.Fraction(instance[0], instance[1])
        if (reduced.numerator != instance[0] or
            reduced.denominator != instance[1]):
            yield ValidationError("Fraction is not in simplified form.")

FractionType.validators = {'simplified': validate_simplified}
```

14.5 Defining custom extension classes

Extension classes are the mechanism that `asdf` uses to register custom tag types so that they can be used when processing ASDF files. Packages that define their own custom tag types must also define extensions in order for those types to be used.

All extension classes must implement the `asdf.AsdfExtension` abstract base class. A custom extension will override each of the following properties of `AsdfExtension` (the text in each bullet is also a link to the corresponding documentation):

- `types`
- `tag_mapping`
- `url_mapping`

14.5.1 Overriding built-in extensions

It is possible for externally defined extensions to override tag types that are provided by asdf's built-in extension. For example, maybe an external package wants to provide a different implementation of NDArrayType. In this case, the external package does not need to provide custom schemas since the schema for the type to be overridden is already provided as part of the ASDF standard.

Instead, the extension class may inherit from asdf's BuiltinExtension and simply override the types property to indicate the type that is being overridden. Doing this preserves the tag_mapping and url_mapping that is used by the BuiltinExtension, which allows the schemas that are packaged by asdf to be located.

asdf will give precedence to the type that is provided by the external extension, effectively overriding the corresponding type in the built-in extension. Note that it is currently undefined if multiple external extensions are provided that override the same built-in type.

14.6 Packaging custom extensions

14.6.1 Packaging schemas

If a package provides custom schemas, the schema files must be installed as part of that package distribution. In general, schema files must be installed into a subdirectory of the package distribution. The asdf extension class must supply a url_mapping that maps to the installed location of the schemas. See *Defining custom extension classes* for more details.

14.6.2 Registering entry points

Packages that provide their own ASDF extensions can (and should!) install them so that they are automatically detectable by the asdf Python package. This is accomplished using Python's setuptools entry points. Entry points are registered in a package's setup.py file.

Consider a package that provides an extension class MyPackageExtension in the submodule mypackage.asdf.extensions. We need to register this class as an extension entry point that asdf will recognize. First, we create a dictionary:

```
entry_points = {}
entry_points['asdf_extensions'] = [
    'mypackage = mypackage.asdf.extensions:MyPackageExtension'
]
```

The key used in the entry_points dictionary must be 'asdf_extensions'. The value must be an array of one or more strings, each with the following format:

```
extension_name = fully.specified.submodule:ExtensionClass
```

The extension name can be any arbitrary string, but it should be descriptive of the package and the extension. In most cases the package itself name will suffice.

Note that depending on individual package requirements, there may be other entries in the entry_points dictionary.

The entry points must be passed to the call to setuptools.setup:

```
from setuptools import setup

entry_points = {}
```

(continues on next page)

(continued from previous page)

```
entry_points['asdf_extensions'] = [
    'mypackage = mypackage.asdf.extensions:MyPackageExtension'
]

setup(
    # We omit other package-specific arguments that are not
    # relevant to this example
    entry_points=entry_points,
)
```

When running `python setup.py install` or `python setup.py develop` on this package, the entry points will be registered automatically. This allows the `asdf` package to recognize the extensions without any user intervention. Users of your package that wish to read ASDF files using types that you have registered will not need to use any extension explicitly. Instead, `asdf` will automatically recognize the types you have registered and will process them appropriately. See *Extensions from other packages* for more information on using extensions.

14.7 Testing custom schemas

Packages that provide their own schemas can test them using `asdf`'s pytest plugin for schema testing. Schemas are tested for overall validity, and any examples given within the schemas are also tested.

The schema tester plugin is automatically registered when the `asdf` package is installed. In order to enable testing, it is necessary to add the directory containing your schema files to the pytest section of your project's build configuration (`pyproject.toml` or `setup.cfg`). If you do not already have such a file, creating one with the following should be sufficient:

```
[tool.pytest.ini_options]
asdf_schema_root = 'path/to/schemas another/path/to/schemas'
```

```
[tool:pytest]
asdf_schema_root = path/to/schemas another/path/to/schemas
```

The schema directory paths should be paths that are relative to the top of the package directory **when it is installed**. If this is different from the path in the source directory, then both paths can be used to facilitate in-place testing (see `asdf`'s own `pyproject.toml` for an example of this).

Note: Older versions of `asdf` (prior to 2.4.0) required the plugin to be registered in your project's `conftest.py` file. As of 2.4.0, the plugin is now registered automatically and so this line should be removed from your `conftest.py` file, unless you need to retain compatibility with older versions of `asdf`.

The `asdf_schema_skip_names` configuration variable can be used to skip schema files that live within one of the `asdf_schema_root` directories but should not be tested. The names should be given as simple base file names (without directory paths or extensions). Again, see `asdf`'s own `pyproject.toml` file for an example.

The schema tests do **not** run by default. In order to enable the tests by default for your package, add `asdf_schema_tests_enabled = 'true'` to the `[tool.pytest.ini_options]` section of your `pyproject.toml` file (or `[tool:pytest]` in `setup.cfg`). If you do not wish to enable the schema tests by default, you can add the `--asdf-tests` option to the `pytest` command line to enable tests on a per-run basis.

Part III

API Documentation

CHAPTER

FIFTEEN

USER API

15.1 asdf Package

asdf: Python library for reading and writing Advanced Scientific Data Format (ASDF) files

15.1.1 Functions

open(fd[, uri, mode, validate_checksums, ...])	Open an existing ASDF file.
info(node_or_path[, max_rows, max_cols, ...])	Print a rendering of an ASDF tree or sub-tree to stdout.
get_config()	Get the current config, which may have been altered by one or more surrounding calls to asdf.config_context.
config_context()	Context manager that temporarily overrides asdf configuration.

open

asdf.**open**(*fd*, *uri=None*, *mode=None*, *validate_checksums=False*, *extensions=None*, *ignore_version_mismatch=True*, *ignore_unrecognized_tag=False*, *_force_raw_types=False*, *copy_arrays=False*, *lazy_load=True*, *custom_schema=None*, *strict_extension_check=False*, *ignore_missing_extensions=False*, *_compat=False*, ***kwargs*)

Open an existing ASDF file.

Parameters

fd

[string or file-like object] May be a string `file` or `http` URI, or a Python file-like object.

uri

[string, optional] The URI of the file. Only required if the URI can not be automatically determined from `fd`.

mode

[string, optional] The mode to open the file in. Must be `r` (default) or `rw`.

validate_checksums

[bool, optional] If `True`, validate the blocks against their checksums. Requires reading the entire file, so disabled by default.

extensions

[object, optional] Additional extensions to use when reading and writing the file. May be any of the following: `asdf.extension.AsdfExtension`, `asdf.extension.Extension`, `asdf.extension.AsdfExtensionList` or a `list` of extensions.

ignore_version_mismatch

[bool, optional] When `True`, do not raise warnings for mismatched schema versions. Set to `True` by default.

ignore_unrecognized_tag

[bool, optional] When `True`, do not raise warnings for unrecognized tags. Set to `False` by default.

copy_arrays

[bool, optional] When `False`, when reading files, attempt to memmap underlying data arrays when possible.

lazy_load

[bool, optional] When `True` and the underlying file handle is seekable, data arrays will only be loaded lazily: i.e. when they are accessed for the first time. In this case the underlying file must stay open during the lifetime of the tree. Setting to False causes all data arrays to be loaded up front, which means that they can be accessed even after the underlying file is closed. Note: even if `lazy_load` is `False`, `copy_arrays` is still taken into account.

custom_schema

[str, optional] Path to a custom schema file that will be used for a secondary validation pass. This can be used to ensure that particular ASDF files follow custom conventions beyond those enforced by the standard.

strict_extension_check

[bool, optional] When `True`, if the given ASDF file contains metadata about the extensions used to create it, and if those extensions are not installed, opening the file will fail. When `False`, opening a file under such conditions will cause only a warning. Defaults to `False`.

ignore_missing_extensions

[bool, optional] When `True`, do not raise warnings when a file is read that contains metadata about extensions that are not available. Defaults to `False`.

validate_on_read

[bool, optional] DEPRECATED. When `True`, validate the newly opened file against tag and custom schemas. Recommended unless the file is already known to be valid.

Returns

asdffile

[AsdfFile] The new AsdfFile object.

info

asdf.**info**(*node_or_path*, *max_rows=24*, *max_cols=120*, *show_values=True*)

Print a rendering of an ASDF tree or sub-tree to stdout.

Parameters

node_or_path

[str, pathlib.Path, asdf.asdf.AsdfFile, or any ASDF tree node] The tree or sub-tree to render. Strings and Path objects will first be passed to asdf.open(...).

max_rows

[int, tuple, or None, optional] Maximum number of lines to print. Nodes that cannot be displayed will be elided with a message. If int, constrain total number of displayed lines. If tuple, constrain lines per node at the depth corresponding to the tuple index. If None, display all lines.

max_cols

[int or None, optional] Maximum length of line to print. Nodes that cannot be fully displayed will be truncated with a message. If int, constrain length of displayed lines. If None, line length is unconstrained.

show_values

[bool, optional] Set to False to disable display of primitive values in the rendered tree.

get_config

asdf.**get_config**()

Get the current config, which may have been altered by one or more surrounding calls to `asdf.config_context`.

Returns

asdf.config.AsdfConfig

config_context

asdf.**config_context**()

Context manager that temporarily overrides asdf configuration. The context yields an asdf.config.AsdfConfig instance that can be modified without affecting code outside of the context.

15.1.2 Classes

AsdfFile([tree, uri, extensions, version, ...])	The main class that represents an ASDF file object.
CustomType()	Base class for all user-defined types.
Stream(shape, dtype[, strides])	Used to put a streamed array into the tree.
IntegerType(value[, storage_type])	Enables the storage of arbitrarily large integer values
ExternalArrayReference(fileuri, target, ...)	Store a reference to an array in an external File.

AsdfFile

class asdf.**AsdfFile**(*tree=None*, *uri=None*, *extensions=None*, *version=None*, *ignore_version_mismatch=True*, *ignore_unrecognized_tag=False*, *ignore_implicit_conversion=False*, *copy_arrays=False*, *lazy_load=True*, *custom_schema=None*, *_readonly=False*)

Bases: object

The main class that represents an ASDF file object.

Parameters

tree

[dict or AsdfFile, optional] The main tree data in the ASDF file. Must conform to the ASDF schema.

uri

[str, optional] The URI for this ASDF file. Used to resolve relative references against. If not provided, will be automatically determined from the associated file object, if possible and if created from AsdfFile.open.

extensions

[object, optional] Additional extensions to use when reading and writing the file. May be any of the following: asdf.extension.AsdfExtension, asdf.extension.Extension, asdf.extension.AsdfExtensionList or a list of extensions.

version

[str, optional] The ASDF Standard version. If not provided, defaults to the configured default version. See asdf.config.AsdfConfig.default_version.

ignore_version_mismatch

[bool, optional] When True, do not raise warnings for mismatched schema versions. Set to True by default.

ignore_unrecognized_tag

[bool, optional] When `True`, do not raise warnings for unrecognized tags. Set to `False` by default.

ignore_implicit_conversion

[bool] When `True`, do not raise warnings when types in the tree are implicitly converted into a serializable object. The motivating case for this is currently `namedtuple`, which cannot be serialized as-is.

copy_arrays

[bool, optional] When `False`, when reading files, attempt to memmap underlying data arrays when possible.

lazy_load

[bool, optional] When `True` and the underlying file handle is seekable, data arrays will only be loaded lazily: i.e. when they are accessed for the first time. In this case the underlying file must stay open during the lifetime of the tree. Setting to False causes all data arrays to be loaded up front, which means that they can be accessed even after the underlying file is closed. Note: even if `lazy_load` is `False`, `copy_arrays` is still taken into account.

custom_schema

[str, optional] Path to a custom schema file that will be used for a secondary validation pass. This can be used to ensure that particular ASDF files follow custom conventions beyond those enforced by the standard.

Attributes Summary

blocks	Get the block manager associated with the AsdfFile.
comments	Get the comments after the header, before the tree.
extension_list	Get the AsdfExtensionList for this AsdfFile.
extension_manager	Get the ExtensionManager for this AsdfFile.
extensions	Get the list of user extensions that are enabled for use with this AsdfFile.
file_format_version	
resolver	
tag_mapping	
tag_to_schema_resolver	
tree	Get/set the tree of data in the ASDF file.
type_index	
uri	Get the URI associated with the AsdfFile.
url_mapping	
version	Get this AsdfFile's ASDF Standard version.
version_map	
version_string	Get this AsdfFile's ASDF Standard version as a string.

Methods Summary

add_history_entry(description[, software])	Add an entry to the history list.
close()	Close the file handles associated with the AsdfFile.
copy()	
fill_defaults()	Fill in any values that are missing in the tree using default values from the schema.
find_references()	Finds all external "JSON References" in the tree and converts them to reference.Reference objects.
get_array_compression(arr)	Get the compression type for the given array data.
get_array_compression_kwargs(arr)	
get_array_storage(arr)	Get the block type for the given array data.
get_history_entries()	Get a list of history entries from the file object.
info([max_rows, max_cols, show_values, ...])	Print a rendering of this file's tree to stdout.
keys()	
make_reference([path])	Make a new reference to a part of this file's tree, that can be assigned as a reference to another tree.
open(fd[, uri, mode, validate_checksums, ...])	Open an existing ASDF file.
open_external(uri, **kwargs)	Open an external ASDF file, from the given (possibly relative) URI.
remove_defaults()	Remove any values in the tree that are the same as the default values in the schema
resolve_and_inline()	Resolves all external references and inlines all data.
resolve_references(**kwargs)	Finds all external "JSON References" in the tree, loads the external content, and places it directly in the tree.
resolve_uri(uri)	Resolve a (possibly relative) URI against the URI of this ASDF file.
run_hook(hookname)	Run a "hook" for each custom type found in the tree.
run_modifying_hook(hookname[, validate])	Run a "hook" for each custom type found in the tree.
search([key, type, value, filter])	Search this file's tree.
set_array_compression(arr, compression, ...)	Set the compression to use for the given array data.
set_array_storage(arr, array_storage)	Set the block type to use for the given array data.
update([all_array_storage, ...])	Update the file on disk in place.
validate()	Validate the current state of the tree against the ASDF schema.
write_to(fd[, all_array_storage, ...])	Write the ASDF file to the given file-like object.

Attributes Documentation

blocks

Get the block manager associated with the AsdfFile.

comments

Get the comments after the header, before the tree.

`extension_list`

Get the AsdfExtensionList for this AsdfFile.

Returns

asdf.extension.AsdfExtensionList

`extension_manager`

Get the ExtensionManager for this AsdfFile.

Returns

asdf.extension.ExtensionManager

`extensions`

Get the list of user extensions that are enabled for use with this AsdfFile.

Returns

list of asdf.extension.ExtensionProxy

`file_format_version`

`resolver`

`tag_mapping`

`tag_to_schema_resolver`

`tree`

Get/set the tree of data in the ASDF file.

When set, the tree will be validated against the ASDF schema.

type_index

uri

Get the URI associated with the `AsdfFile`.

In many cases, it is automatically determined from the file handle used to read or write the file.

url_mapping

version

Get this AsdfFile's ASDF Standard version.

Returns

asdf.versioning.AsdfVersion

version_map

version_string

Get this AsdfFile's ASDF Standard version as a string.

Returns

str

Methods Documentation

add_history_entry(*description*, *software=None*)

Add an entry to the history list.

Parameters

description

[str] A description of the change.

software

[dict or list of dict] A description of the software used. It should not include asdf itself, as that is automatically notated in the `asdf_library` entry.

Each dict must have the following keys:

- `name`: The name of the software
- `author`: The author or institution that produced the software
- `homepage`: A URI to the homepage of the software
- `version`: The version of the software

close()

Close the file handles associated with the `AsdfFile`.

copy()

fill_defaults()

Fill in any values that are missing in the tree using default values from the schema.

find_references()

Finds all external "JSON References" in the tree and converts them to `reference.Reference` objects.

get_array_compression(*arr*)

Get the compression type for the given array data.

Parameters

arr

[numpy.ndarray]

Returns

compression

[str or None]

get_array_compression_kwargs(*arr*)

get_array_storage(*arr*)

Get the block type for the given array data.

Parameters

arr

[numpy.ndarray]

`get_history_entries()`

Get a list of history entries from the file object.

Returns

entries

[list] A list of history entries.

`info(max_rows=24, max_cols=120, show_values=True, refresh_extension_manager=False)`

Print a rendering of this file's tree to stdout.

Parameters

max_rows

[int, tuple, or None, optional] Maximum number of lines to print. Nodes that cannot be displayed will be elided with a message. If int, constrain total number of displayed lines. If tuple, constrain lines per node at the depth corresponding to the tuple index. If None, display all lines.

max_cols

[int or None, optional] Maximum length of line to print. Nodes that cannot be fully displayed will be truncated with a message. If int, constrain length of displayed lines. If None, line length is unconstrained.

show_values

[bool, optional] Set to False to disable display of primitive values in the rendered tree.

`keys()` → a set-like object providing a view on D's keys

`make_reference(path=[])`

Make a new reference to a part of this file's tree, that can be assigned as a reference to another tree.

Parameters

path

[list of str and int, optional] The parts of the path pointing to an item in this tree. If omitted, points to the root of the tree.

Returns

reference

[reference.Reference] A reference object.

Examples

For the given AsdfFile `ff`, add an external reference to the data in an external file:

```
>>> import asdf
>>> flat = asdf.open("http://stsci.edu/reference_files/flat.asdf")
>>> ff.tree['flat_field'] = flat.make_reference(['data'])
```

classmethod open(*fd*, *uri=None*, *mode='r'*, *validate_checksums=False*, *extensions=None*, *ignore_version_mismatch=True*, *ignore_unrecognized_tag=False*, *_force_raw_types=False*, *copy_arrays=False*, *lazy_load=True*, *custom_schema=None*, *strict_extension_check=False*, *ignore_missing_extensions=False*, ***kwargs*)

Open an existing ASDF file.

Deprecated since version 2.2: Use `asdf.open` instead.

open_external(*uri*, ***kwargs*)

Open an external ASDF file, from the given (possibly relative) URI. There is a cache (internal to this ASDF file) that ensures each external ASDF file is loaded only once.

Parameters

uri

[str] An absolute or relative URI to resolve against the URI of this ASDF file.

Returns

asdffile

[AsdfFile] The external ASDF file.

remove_defaults()

Remove any values in the tree that are the same as the default values in the schema

resolve_and_inline()

Resolves all external references and inlines all data. This produces something that, when saved, is a 100% valid YAML file.

resolve_references(***kwargs*)

Finds all external "JSON References" in the tree, loads the external content, and places it directly in

the tree. Saving a ASDF file after this operation means it will have no external references, and will be completely self-contained.

`resolve_uri`(*uri*)

Resolve a (possibly relative) URI against the URI of this ASDF file. May be overridden by base classes to change how URIs are resolved. This does not apply any `uri_mapping` that was passed to the constructor.

Parameters

uri

[str] An absolute or relative URI to resolve against the URI of this ASDF file.

Returns

uri

[str] The resolved URI.

`run_hook`(*hookname*)

Run a "hook" for each custom type found in the tree.

Parameters

hookname

[str] The name of the hook. If a `AsdfType` is found with a method with this name, it will be called for every instance of the corresponding custom type in the tree.

`run_modifying_hook`(*hookname*, *validate=True*)

Run a "hook" for each custom type found in the tree. The hook is free to return a different object in order to modify the tree.

Parameters

hookname

[str] The name of the hook. If a `AsdfType` is found with a method with this name, it will be called for every instance of the corresponding custom type in the tree.

validate

[bool] When `True` (default) validate the resulting tree.

`search`(*key=NotSet*, *type=NotSet*, *value=NotSet*, *filter=None*)

Search this file's tree.

Parameters

key

[NotSet, str, or any other object] Search query that selects nodes by dict key or list index. If NotSet, the node key is unconstrained. If str, the input is searched among keys/indexes as a regular expression pattern. If any other object, node's key or index must equal the queried key.

type

[NotSet, str, or builtins.type] Search query that selects nodes by type. If NotSet, the node type is unconstrained. If str, the input is searched among (fully qualified) node type names as a regular expression pattern. If builtins.type, the node must be an instance of the input.

value

[NotSet, str, or any other object] Search query that selects nodes by value. If NotSet, the node value is unconstrained. If str, the input is searched among values as a regular expression pattern. If any other object, node's value must equal the queried value.

filter

[callable] Callable that filters nodes by arbitrary criteria. The callable accepts one or two arguments:

- the node
- the node's list index or dict key (optional)

and returns True to retain the node, or False to remove it from the search results.

Returns

asdf.search.AsdfSearchResult

the result of the search

`set_array_compression`(*arr*, *compression*, ***compression_kwargs*)

Set the compression to use for the given array data.

Parameters

arr

[numpy.ndarray] The array to set. If multiple views of the array are in the tree, only the most recent compression setting will be used, since all views share a single block.

compression

[str or None] Must be one of:

- `''` or `None`: no compression

- `zlib`: Use zlib compression
- `bzp2`: Use bzip2 compression
- `lz4`: Use lz4 compression
- `input`: Use the same compression as in the file read. If there is no prior file, acts as None.

set_array_storage(*arr*, *array_storage*)

Set the block type to use for the given array data.

Parameters

arr

[numpy.ndarray] The array to set. If multiple views of the array are in the tree, only the most recent block type setting will be used, since all views share a single block.

array_storage

[str] Must be one of:

- `internal`: The default. The array data will be stored in a binary block in the same ASDF file.
- `external`: Store the data in a binary block in a separate ASDF file.
- `inline`: Store the data as YAML inline in the tree.

update(*all_array_storage=None*, *all_array_compression='input'*, *pad_blocks=False*, *include_block_index=True*, *version=None*, *compression_kwargs=None*, ***kwargs*)

Update the file on disk in place.

Parameters

all_array_storage

[string, optional] If provided, override the array storage type of all blocks in the file immediately before writing. Must be one of:

- `internal`: The default. The array data will be stored in a binary block in the same ASDF file.
- `external`: Store the data in a binary block in a separate ASDF file.
- `inline`: Store the data as YAML inline in the tree.

all_array_compression

[string, optional] If provided, set the compression type on all binary blocks in the file. Must be one of:

- `''` or `None`: No compression.
- `zlib`: Use zlib compression.
- `bzp2`: Use bzip2 compression.

- `lz4`: Use lz4 compression.
- `input`: Use the same compression as in the file read. If there is no prior file, acts as None

pad_blocks

[float or bool, optional] Add extra space between blocks to allow for updating of the file. If `False` (default), add no padding (always return 0). If `True`, add a default amount of padding of 10% If a float, it is a factor to multiple content_size by to get the new total size.

include_block_index

[bool, optional] If `False`, don't include a block index at the end of the file. (Default: `True`) A block index is never written if the file has a streamed block.

version

[str, optional] Update the ASDF Standard version of this AsdfFile before writing.

auto_inline

[int, optional] DEPRECATED. When the number of elements in an array is less than this threshold, store the array as inline YAML, rather than a binary block. This only works on arrays that do not share data with other arrays. Default is the value specified in `asdf.get_config().array_inline_threshold`.

validate()

Validate the current state of the tree against the ASDF schema.

write_to(*fd*, *all_array_storage=None*, *all_array_compression='input'*, *pad_blocks=False*, *include_block_index=True*, *version=None*, *compression_kwargs=None*, ***kwargs*)

Write the ASDF file to the given file-like object.

`write_to` does not change the underlying file descriptor in the `AsdfFile` object, but merely copies the content to a new file.

Parameters

fd

[string or file-like object] May be a string path to a file, or a Python file-like object. If a string path, the file is automatically closed after writing. If not a string path, it is the caller's responsibility to close the object.

all_array_storage

[string, optional] If provided, override the array storage type of all blocks in the file immediately before writing. Must be one of:

- `internal`: The default. The array data will be stored in a binary block in the same ASDF file.
- `external`: Store the data in a binary block in a separate ASDF file.
- `inline`: Store the data as YAML inline in the tree.

all_array_compression

[string, optional] If provided, set the compression type on all binary blocks in the file. Must be one of:

- `''` or `None`: No compression.
- `zlib`: Use zlib compression.
- `bzp2`: Use bzip2 compression.
- `lz4`: Use lz4 compression.
- `input`: Use the same compression as in the file read. If there is no prior file, acts as None.

pad_blocks

[float or bool, optional] Add extra space between blocks to allow for updating of the file. If `False` (default), add no padding (always return 0). If `True`, add a default amount of padding of 10% If a float, it is a factor to multiple content_size by to get the new total size.

include_block_index

[bool, optional] If `False`, don't include a block index at the end of the file. (Default: `True`) A block index is never written if the file has a streamed block.

version

[str, optional] Update the ASDF Standard version of this AsdfFile before writing.

auto_inline

[int, optional] DEPRECATED. When the number of elements in an array is less than this threshold, store the array as inline YAML, rather than a binary block. This only works on arrays that do not share data with other arrays. Default is the value specified in `asdf.get_config().array_inline_threshold`.

CustomType

class asdf.**CustomType**

Bases: `ExtensionType`

Base class for all user-defined types.

Attributes Summary

handle_dynamic_subclasses	bool: Indicates whether dynamically generated subclasses can be serialized
has_required_modules	bool: Indicates whether modules specified by requires are available.
name	str or list: The name of the type.
organization	str: The organization responsible for the type.
requires	list: Python packages that are required to instantiate the object.
standard	str: The standard the type is defined in.
supported_versions	set: Versions that explicitly compatible with this extension class.
types	list: List of types that this extension class can convert to/from YAML.
validators	dict: Mapping JSON Schema keywords to validation functions for jsonschema.
version	str, tuple, AsdfVersion, or AsdfSpec: The version of the type.
yaml_tag	str: The YAML tag to use for the type.

Methods Summary

from_tree(tree, ctx)	Converts basic types representing YAML trees into custom types.
from_tree_tagged(tree, ctx)	Converts from tagged tree into custom type.
incompatible_version(version)	Indicates if given version is known to be incompatible with this type.
make_yaml_tag(name[, versioned])	Given the name of a type, returns a string representing its YAML tag.
names()	Returns the name(s) represented by this tag type as a list.
tag_base()	Returns the base of the YAML tag for types represented by this class.
to_tree(node, ctx)	Converts instances of custom types into YAML representations.
to_tree_tagged(node, ctx)	Converts instances of custom types into tagged objects.

Attributes Documentation

handle_dynamic_subclasses = False

bool: Indicates whether dynamically generated subclasses can be serialized

Flag indicating whether this type is capable of serializing subclasses of any of the types listed in types that are generated dynamically.

has_required_modules = True

bool: Indicates whether modules specified by requires are available.

NOTE: This value is automatically generated. Do not set it in subclasses as it will be overwritten.

name = None

str or list: The name of the type.

organization = 'stsci.edu'

str: The organization responsible for the type.

requires = []

list: Python packages that are required to instantiate the object.

standard = 'asdf'

str: The standard the type is defined in.

supported_versions = []

set: Versions that explicitly compatible with this extension class.

If provided, indicates explicit compatibility with the given set of versions. Other versions of the same schema that are not included in this set will not be converted to custom types with this class.

types = []

list: List of types that this extension class can convert to/from YAML.

Custom Python types that, when found in the tree, will be converted into basic types for YAML output. Can be either strings referring to the types or the types themselves.

validators = {}

dict: Mapping JSON Schema keywords to validation functions for jsonschema.

Useful if the type defines extra types of validation that can be performed.

version = AsdfVersion('1.0.0')

str, tuple, AsdfVersion, or AsdfSpec: The version of the type.

yaml_tag = None

str: The YAML tag to use for the type.

If not provided, it will be automatically generated from name, organization, standard and version.

Methods Documentation

classmethod from_tree(*tree*, *ctx*)

Converts basic types representing YAML trees into custom types.

This method should be overridden by custom extension classes in order to define how custom types are deserialized from the YAML representation back into their original types. Typically the method will return an instance of the original custom type. It is also permitted to return a generator, which yields a partially constructed result, then completes construction once the generator is drained. This is useful when constructing objects that contain reference cycles.

This method is called as part of the process of reading an ASDF file in order to construct an `AsdfFile` object. Whenever a YAML subtree is encountered that has a tag that corresponds to the `yaml_tag` property of this class, this method will be used to deserialize that tree back into an instance of the original custom type.

Parameters

tree

[`object` representing YAML tree] An instance of a basic Python type (possibly nested) that corresponds to a YAML subtree.

ctx

[`AsdfFile`] An instance of the `AsdfFile` object that is being constructed.

Returns

An instance of the custom type represented by this extension class, or a generator that yields that instance.

classmethod from_tree_tagged(*tree*, *ctx*)

Converts from tagged tree into custom type.

It is more common for extension classes to override `from_tree` instead of this method. This method should only be overridden if it is necessary to access the `_tag` property of the `Tagged` object directly.

Parameters

tree

[`asdf.tagged.Tagged` object representing YAML tree]

ctx

[`AsdfFile`] An instance of the `AsdfFile` object that is being constructed.

Returns

An instance of the custom type represented by this extension class.

classmethod incompatible_version(*version*)

Indicates if given version is known to be incompatible with this type.

If this tag class explicitly identifies compatible versions then this checks whether a given version is compatible or not (see `supported_versions`). Otherwise, all versions are assumed to be compatible.

Child classes can override this method to affect how version compatiblity for this type is determined.

Parameters

version

[`str` or `AsdfVersion`] The version to test for compatibility.

classmethod make_yaml_tag(*name*, *versioned=True*)

Given the name of a type, returns a string representing its YAML tag.

Parameters

name

[str] The name of the type. In most cases this will correspond to the `name` attribute of the tag type. However, it is passed as a parameter since some tag types represent multiple custom types.

versioned

[bool] If `True`, the tag will be versioned. Otherwise, a YAML tag without a version will be returned.

Returns

`str` representing the YAML tag

classmethod names()

Returns the name(s) represented by this tag type as a list.

While some tag types represent only a single custom type, others represent multiple types. In the latter case, the `name` attribute of the extension is actually a list, not simply a string. This method normalizes the value of `name` by returning a list in all cases.

Returns

list of names represented by this tag type

classmethod tag_base()

Returns the base of the YAML tag for types represented by this class.

This method returns the portion of the tag that represents the standard and the organization of any type represented by this class.

Returns

str representing the base of the YAML tag

classmethod to_tree(*node*, *ctx*)

Converts instances of custom types into YAML representations.

This method should be overridden by custom extension classes in order to define how custom types are serialized into YAML. The method must return a single Python object corresponding to one of the basic YAML types (dict, list, str, or number). However, the types can be nested and combined in order to represent more complex custom types.

This method is called as part of the process of writing an AsdfFile object. Whenever a custom type (or a subclass of that type) that is listed in the types attribute of this class is encountered, this method will be used to serialize that type.

The name to_tree refers to the act of converting a custom type into part of a YAML object tree.

Parameters

node

[object] Instance of a custom type to be serialized. Will be an instance (or an instance of a subclass) of one of the types listed in the types attribute of this class.

ctx

[AsdfFile] An instance of the AsdfFile object that is being written out.

Returns

A basic YAML type (dict, list, str, int, float, or complex) representing the properties of the custom type to be serialized. These types can be nested in order to represent more complex custom types.

classmethod to_tree_tagged(*node*, *ctx*)

Converts instances of custom types into tagged objects.

It is more common for custom tag types to override `to_tree` instead of this method. This method should be overridden if it is necessary to modify the YAML tag that will be used to tag this object.

Parameters

node

[`object`] Instance of a custom type to be serialized. Will be an instance (or an instance of a subclass) of one of the types listed in the `types` attribute of this class.

ctx

[`AsdfFile`] An instance of the `AsdfFile` object that is being written out.

Returns

An instance of `asdf.tagged.Tagged`.

Stream

class asdf.**Stream**(*shape*, *dtype*, *strides=None*)

Bases: `NDArrayType`

Used to put a streamed array into the tree.

Examples

Save a double-precision array with 1024 columns, one row at a time:

```
>>> from asdf import AsdfFile, Stream
>>> import numpy as np
>>> ff = AsdfFile()
>>> ff.tree['streamed'] = Stream([1024], np.float64)
>>> with open('test.asdf', 'wb') as fd:
...     ff.write_to(fd)
...     for i in range(200):
...         nbytes = fd.write(
...                     np.array([i] * 1024, np.float64).tobytes())
```

Attributes Summary

block
dtype
handle_dynamic_subclasses
has_required_modules
name
organization
requires
shape
standard
supported_versions
types
validators
version
yaml_tag

Methods Summary

assert_allclose(old, new)	
assert_equal(old, new)	
copy_to_new_asdf(node, asdffile)	
from_tree(data, ctx)	Converts basic types representing YAML trees into custom types.
from_tree_tagged(tree, ctx)	Converts from tagged tree into custom type.
get_actual_shape(shape, strides, dtype, ...)	Get the actual shape of an array, by computing it against the block_size if it contains a *.
incompatible_version(version)	Indicates if given version is known to be incompatible with this type.
make_yaml_tag(name[, versioned])	Given the name of a type, returns a string representing its YAML tag.
names()	Returns the name(s) represented by this tag type as a list.
reserve_blocks(data, ctx)	
tag_base()	Returns the base of the YAML tag for types represented by this class.
to_tree(data, ctx)	Converts instances of custom types into YAML representations.
to_tree_tagged(node, ctx)	Converts instances of custom types into tagged objects.

Attributes Documentation

block

dtype

handle_dynamic_subclasses = False

has_required_modules = True

name = None

organization = 'stsci.edu'

requires = []

shape

standard = 'asdf'

supported_versions = []

types = []

validators = {'datatype': <function validate_datatype>, 'max_ndim': <function validate_max_ndim>, 'ndim': <function validate_ndim>}

version = AsdfVersion('1.0.0')

yaml_tag = 'tag:stsci.edu:asdf/core/ndarray-1.0.0'

Methods Documentation

classmethod assert_allclose(*old*, *new*)

classmethod assert_equal(*old*, *new*)

classmethod copy_to_new_asdf(*node*, *asdffile*)

classmethod from_tree(*data*, *ctx*)

Converts basic types representing YAML trees into custom types.

This method should be overridden by custom extension classes in order to define how custom types are deserialized from the YAML representation back into their original types. Typically the method will return an instance of the original custom type. It is also permitted to return a generator, which yields a partially constructed result, then completes construction once the generator is drained. This is useful when constructing objects that contain reference cycles.

This method is called as part of the process of reading an ASDF file in order to construct an `AsdfFile` object. Whenever a YAML subtree is encountered that has a tag that corresponds to the `yaml_tag` property of this class, this method will be used to deserialize that tree back into an instance of the original custom type.

Parameters

tree

[`object` representing YAML tree] An instance of a basic Python type (possibly nested) that corresponds to a YAML subtree.

ctx

[`AsdfFile`] An instance of the `AsdfFile` object that is being constructed.

Returns

An instance of the custom type represented by this extension class, or a generator that yields that instance.

classmethod from_tree_tagged(*tree*, *ctx*)

Converts from tagged tree into custom type.

It is more common for extension classes to override `from_tree` instead of this method. This method should only be overridden if it is necessary to access the `_tag` property of the `Tagged` object directly.

Parameters

tree

[`asdf.tagged.Tagged` object representing YAML tree]

ctx

[`AsdfFile`] An instance of the `AsdfFile` object that is being constructed.

Returns

An instance of the custom type represented by this extension class.

get_actual_shape(*shape*, *strides*, *dtype*, *block_size*)

Get the actual shape of an array, by computing it against the block_size if it contains a `*`.

classmethod incompatible_version(*version*)

Indicates if given version is known to be incompatible with this type.

If this tag class explicitly identifies compatible versions then this checks whether a given version is compatible or not (see `supported_versions`). Otherwise, all versions are assumed to be compatible.

Child classes can override this method to affect how version compatiblity for this type is determined.

Parameters

version

[`str` or `AsdfVersion`] The version to test for compatibility.

classmethod make_yaml_tag(*name*, *versioned=True*)

Given the name of a type, returns a string representing its YAML tag.

Parameters

name

[str] The name of the type. In most cases this will correspond to the `name` attribute of the tag type. However, it is passed as a parameter since some tag types represent multiple custom types.

versioned

[bool] If `True`, the tag will be versioned. Otherwise, a YAML tag without a version will be returned.

Returns

`str` representing the YAML tag

classmethod names()

Returns the name(s) represented by this tag type as a list.

While some tag types represent only a single custom type, others represent multiple types. In the latter case, the `name` attribute of the extension is actually a list, not simply a string. This method normalizes the value of `name` by returning a list in all cases.

Returns

`list` of names represented by this tag type

classmethod reserve_blocks(*data*, *ctx*)

classmethod tag_base()

Returns the base of the YAML tag for types represented by this class.

This method returns the portion of the tag that represents the standard and the organization of any type represented by this class.

Returns

str representing the base of the YAML tag

classmethod to_tree(*data*, *ctx*)

Converts instances of custom types into YAML representations.

This method should be overridden by custom extension classes in order to define how custom types are serialized into YAML. The method must return a single Python object corresponding to one of the basic YAML types (dict, list, str, or number). However, the types can be nested and combined in order to represent more complex custom types.

This method is called as part of the process of writing an AsdfFile object. Whenever a custom type (or a subclass of that type) that is listed in the types attribute of this class is encountered, this method will be used to serialize that type.

The name to_tree refers to the act of converting a custom type into part of a YAML object tree.

Parameters

node

[object] Instance of a custom type to be serialized. Will be an instance (or an instance of a subclass) of one of the types listed in the types attribute of this class.

ctx

[AsdfFile] An instance of the AsdfFile object that is being written out.

Returns

A basic YAML type (dict, list, str, int, float, or complex) representing the properties of the custom type to be serialized. These types can be nested in order to represent more complex custom types.

classmethod to_tree_tagged(*node*, *ctx*)

Converts instances of custom types into tagged objects.

It is more common for custom tag types to override to_tree instead of this method. This method should be overridden if it is necessary to modify the YAML tag that will be used to tag this object.

Parameters

node

[object] Instance of a custom type to be serialized. Will be an instance (or an instance of a subclass) of one of the types listed in the types attribute of this class.

ctx

[AsdfFile] An instance of the AsdfFile object that is being written out.

Returns

An instance of asdf.tagged.Tagged.

IntegerType

class asdf.**IntegerType**(*value*, *storage_type='internal'*)

Bases: AsdfType

Enables the storage of arbitrarily large integer values

The ASDF Standard mandates that integer literals in the tree can be no larger than 64 bits. Use of this class enables the storage of arbitrarily large integer values.

When reading files that contain arbitrarily large integers, the values that are restored in the tree will be raw Python int instances.

Parameters

value: `numbers.Integral`

A Python integral value (e.g. int or numpy.integer)

storage_type: `str`, optional

Optionally overrides the storage type of the array used to represent the integer value. Valid values are "internal" (the default) and "inline"

Examples

```
>>> import asdf
>>> import random
>>> # Create a large integer value
>>> largeval = random.getrandbits(100)
>>> # Store the large integer value to the tree using asdf.IntegerType
>>> tree = dict(largeval=asdf.IntegerType(largeval))
>>> with asdf.AsdfFile(tree) as af:
...     af.write_to('largeval.asdf')
>>> with asdf.open('largeval.asdf') as aa:
...     assert aa['largeval'] == largeval
```

Attributes Summary

handle_dynamic_subclasses
has_required_modules
name
organization
requires
standard
supported_versions
types
validators
version
yaml_tag

Methods Summary

from_tree(tree, ctx)	Converts basic types representing YAML trees into custom types.
from_tree_tagged(tree, ctx)	Converts from tagged tree into custom type.
incompatible_version(version)	Indicates if given version is known to be incompatible with this type.
make_yaml_tag(name[, versioned])	Given the name of a type, returns a string representing its YAML tag.
names()	Returns the name(s) represented by this tag type as a list.
tag_base()	Returns the base of the YAML tag for types represented by this class.
to_tree(node, ctx)	Converts instances of custom types into YAML representations.
to_tree_tagged(node, ctx)	Converts instances of custom types into tagged objects.

Attributes Documentation

handle_dynamic_subclasses = False

has_required_modules = True

name = 'core/integer'

organization = 'stsci.edu'

requires = []

standard = 'asdf'

supported_versions = []

types = []

```
validators = {}
```

```
version = AsdfVersion('1.0.0')
```

```
yaml_tag = 'tag:stsci.edu:asdf/core/integer-1.0.0'
```

Methods Documentation

classmethod from_tree(*tree*, *ctx*)

Converts basic types representing YAML trees into custom types.

This method should be overridden by custom extension classes in order to define how custom types are deserialized from the YAML representation back into their original types. Typically the method will return an instance of the original custom type. It is also permitted to return a generator, which yields a partially constructed result, then completes construction once the generator is drained. This is useful when constructing objects that contain reference cycles.

This method is called as part of the process of reading an ASDF file in order to construct an `AsdfFile` object. Whenever a YAML subtree is encountered that has a tag that corresponds to the `yaml_tag` property of this class, this method will be used to deserialize that tree back into an instance of the original custom type.

Parameters

tree

[`object` representing YAML tree] An instance of a basic Python type (possibly nested) that corresponds to a YAML subtree.

ctx

[`AsdfFile`] An instance of the `AsdfFile` object that is being constructed.

Returns

An instance of the custom type represented by this extension class, or a generator that yields that instance.

classmethod from_tree_tagged(*tree*, *ctx*)

Converts from tagged tree into custom type.

It is more common for extension classes to override `from_tree` instead of this method. This method should only be overridden if it is necessary to access the `_tag` property of the `Tagged` object directly.

Parameters

tree

[asdf.tagged.Tagged object representing YAML tree]

ctx

[AsdfFile] An instance of the AsdfFile object that is being constructed.

Returns

An instance of the custom type represented by this extension class.

classmethod incompatible_version(*version*)

Indicates if given version is known to be incompatible with this type.

If this tag class explicitly identifies compatible versions then this checks whether a given version is compatible or not (see supported_versions). Otherwise, all versions are assumed to be compatible.

Child classes can override this method to affect how version compatiblity for this type is determined.

Parameters

version

[str or AsdfVersion] The version to test for compatibility.

classmethod make_yaml_tag(*name*, *versioned=True*)

Given the name of a type, returns a string representing its YAML tag.

Parameters

name

[str] The name of the type. In most cases this will correspond to the name attribute of the tag type. However, it is passed as a parameter since some tag types represent multiple custom types.

versioned

[bool] If True, the tag will be versioned. Otherwise, a YAML tag without a version will be returned.

Returns

str representing the YAML tag

classmethod names()

Returns the name(s) represented by this tag type as a list.

While some tag types represent only a single custom type, others represent multiple types. In the latter case, the name attribute of the extension is actually a list, not simply a string. This method normalizes the value of name by returning a list in all cases.

Returns

list of names represented by this tag type

classmethod tag_base()

Returns the base of the YAML tag for types represented by this class.

This method returns the portion of the tag that represents the standard and the organization of any type represented by this class.

Returns

str representing the base of the YAML tag

classmethod to_tree(*node*, *ctx*)

Converts instances of custom types into YAML representations.

This method should be overridden by custom extension classes in order to define how custom types are serialized into YAML. The method must return a single Python object corresponding to one of the basic YAML types (dict, list, str, or number). However, the types can be nested and combined in order to represent more complex custom types.

This method is called as part of the process of writing an AsdfFile object. Whenever a custom type (or a subclass of that type) that is listed in the types attribute of this class is encountered, this method will be used to serialize that type.

The name to_tree refers to the act of converting a custom type into part of a YAML object tree.

Parameters

node

[object] Instance of a custom type to be serialized. Will be an instance (or an instance of a subclass) of one of the types listed in the types attribute of this class.

ctx

[AsdfFile] An instance of the AsdfFile object that is being written out.

Returns

A basic YAML type (dict, list, str, int, float, or complex) representing the properties of the custom type to be serialized. These types can be nested in order to represent more complex custom types.

classmethod to_tree_tagged(*node*, *ctx*)

Converts instances of custom types into tagged objects.

It is more common for custom tag types to override to_tree instead of this method. This method should be overridden if it is necessary to modify the YAML tag that will be used to tag this object.

Parameters

node

[object] Instance of a custom type to be serialized. Will be an instance (or an instance of a subclass) of one of the types listed in the types attribute of this class.

ctx

[AsdfFile] An instance of the AsdfFile object that is being written out.

Returns

An instance of asdf.tagged.Tagged.

ExternalArrayReference

class asdf.**ExternalArrayReference**(*fileuri*, *target*, *dtype*, *shape*)

Bases: AsdfType

Store a reference to an array in an external File.

This class is a simple way of referring to an array in another file. It provides no way to resolve these references, that is left to the user. It also performs no checking to see if any of the arguments are correct. e.g. if the file exits.

Parameters

fileuri: `str`

The path to the path to be referenced. Can be relative to the file containing the reference.

target: `object`

Some internal target to the data in the file. Examples may include a HDU index, a HDF path or an asdf fragment.

dtype: `str`

The (numpy) dtype of the contained array.

shape: `tuple`

The shape of the array to be loaded.

Examples

```
>>> import asdf
>>> ref = asdf.ExternalArrayReference("myfitsfile.fits", 1, "float64", (100, 100))
>>> tree = {'reference': ref}
>>> with asdf.AsdfFile(tree) as ff:
...     ff.write_to("test.asdf")
```

Attributes Summary

handle_dynamic_subclasses
has_required_modules
name
organization
requires
standard
supported_versions
types
validators
version
yaml_tag

Methods Summary

from_tree(tree, ctx)	Converts basic types representing YAML trees into custom types.
from_tree_tagged(tree, ctx)	Converts from tagged tree into custom type.
incompatible_version(version)	Indicates if given version is known to be incompatible with this type.
make_yaml_tag(name[, versioned])	Given the name of a type, returns a string representing its YAML tag.
names()	Returns the name(s) represented by this tag type as a list.
tag_base()	Returns the base of the YAML tag for types represented by this class.
to_tree(data, ctx)	Converts instances of custom types into YAML representations.
to_tree_tagged(node, ctx)	Converts instances of custom types into tagged objects.

Attributes Documentation

handle_dynamic_subclasses = False

has_required_modules = True

name = 'core/externalarray'

organization = 'stsci.edu'

requires = []

standard = 'asdf'

supported_versions = []

types = []

```
validators = {}
```

```
version = AsdfVersion('1.0.0')
```

```
yaml_tag = 'tag:stsci.edu:asdf/core/externalarray-1.0.0'
```

Methods Documentation

classmethod from_tree(*tree*, *ctx*)

Converts basic types representing YAML trees into custom types.

This method should be overridden by custom extension classes in order to define how custom types are deserialized from the YAML representation back into their original types. Typically the method will return an instance of the original custom type. It is also permitted to return a generator, which yields a partially constructed result, then completes construction once the generator is drained. This is useful when constructing objects that contain reference cycles.

This method is called as part of the process of reading an ASDF file in order to construct an `AsdfFile` object. Whenever a YAML subtree is encountered that has a tag that corresponds to the `yaml_tag` property of this class, this method will be used to deserialize that tree back into an instance of the original custom type.

Parameters

tree

[`object` representing YAML tree] An instance of a basic Python type (possibly nested) that corresponds to a YAML subtree.

ctx

[`AsdfFile`] An instance of the `AsdfFile` object that is being constructed.

Returns

An instance of the custom type represented by this extension class, or a generator that yields that instance.

classmethod from_tree_tagged(*tree*, *ctx*)

Converts from tagged tree into custom type.

It is more common for extension classes to override `from_tree` instead of this method. This method should only be overridden if it is necessary to access the `_tag` property of the `Tagged` object directly.

Parameters

tree

[asdf.tagged.Tagged object representing YAML tree]

ctx

[AsdfFile] An instance of the AsdfFile object that is being constructed.

Returns

An instance of the custom type represented by this extension class.

classmethod incompatible_version(*version*)

Indicates if given version is known to be incompatible with this type.

If this tag class explicitly identifies compatible versions then this checks whether a given version is compatible or not (see supported_versions). Otherwise, all versions are assumed to be compatible.

Child classes can override this method to affect how version compatiblity for this type is determined.

Parameters

version

[str or AsdfVersion] The version to test for compatibility.

classmethod make_yaml_tag(*name*, *versioned=True*)

Given the name of a type, returns a string representing its YAML tag.

Parameters

name

[str] The name of the type. In most cases this will correspond to the name attribute of the tag type. However, it is passed as a parameter since some tag types represent multiple custom types.

versioned

[bool] If True, the tag will be versioned. Otherwise, a YAML tag without a version will be returned.

Returns

str representing the YAML tag

classmethod names()

Returns the name(s) represented by this tag type as a list.

While some tag types represent only a single custom type, others represent multiple types. In the latter case, the name attribute of the extension is actually a list, not simply a string. This method normalizes the value of name by returning a list in all cases.

Returns

list of names represented by this tag type

classmethod tag_base()

Returns the base of the YAML tag for types represented by this class.

This method returns the portion of the tag that represents the standard and the organization of any type represented by this class.

Returns

str representing the base of the YAML tag

classmethod to_tree(*data*, *ctx*)

Converts instances of custom types into YAML representations.

This method should be overridden by custom extension classes in order to define how custom types are serialized into YAML. The method must return a single Python object corresponding to one of the basic YAML types (dict, list, str, or number). However, the types can be nested and combined in order to represent more complex custom types.

This method is called as part of the process of writing an AsdfFile object. Whenever a custom type (or a subclass of that type) that is listed in the types attribute of this class is encountered, this method will be used to serialize that type.

The name to_tree refers to the act of converting a custom type into part of a YAML object tree.

Parameters

node

[object] Instance of a custom type to be serialized. Will be an instance (or an instance of a subclass) of one of the types listed in the types attribute of this class.

ctx

[AsdfFile] An instance of the AsdfFile object that is being written out.

Returns

A basic YAML type (dict, list, str, int, float, or complex) representing the properties of the custom type to be serialized. These types can be nested in order to represent more complex custom types.

classmethod to_tree_tagged(*node*, *ctx*)

Converts instances of custom types into tagged objects.

It is more common for custom tag types to override to_tree instead of this method. This method should be overridden if it is necessary to modify the YAML tag that will be used to tag this object.

Parameters

node

[object] Instance of a custom type to be serialized. Will be an instance (or an instance of a subclass) of one of the types listed in the types attribute of this class.

ctx

[AsdfFile] An instance of the AsdfFile object that is being written out.

Returns

An instance of asdf.tagged.Tagged.

15.1.3 Variables

__version__	str(object='') -> str str(bytes_or_buffer[, encoding[, errors]]) -> str

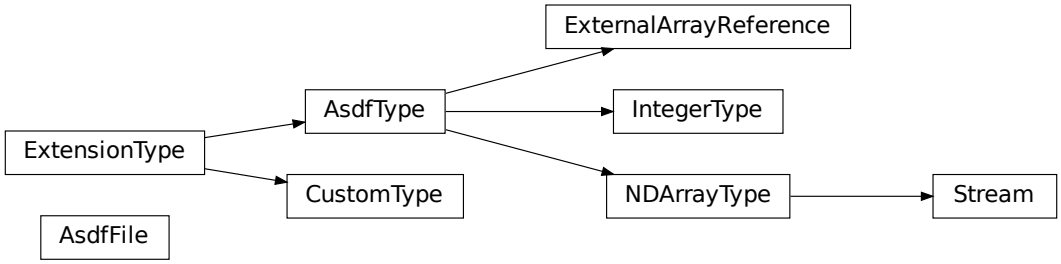
__version__

asdf.**__version__ = '2.12.1.dev90+g784740b'**

str(object='') -> str str(bytes_or_buffer[, encoding[, errors]]) -> str

Create a new string object from the given object. If encoding or errors is specified, then the object must expose a data buffer that will be decoded using the given encoding and error handler. Otherwise, returns the result of object.__str__() (if defined) or repr(object). encoding defaults to sys.getdefaultencoding(). errors defaults to 'strict'.

15.1.4 Class Inheritance Diagram



15.2 asdf.search Module

Utilities for searching ASDF trees.

15.2.1 Classes

AsdfSearchResult(identifiers, node[, ...])	Result of a call to AsdfFile.search.

AsdfSearchResult

class asdf.search.**AsdfSearchResult**(*identifiers*, *node*, *filters=[]*, *parent_node=None*, *max_rows=24*, *max_cols=120*, *show_values=True*)

Bases: object

Result of a call to AsdfFile.search.

Attributes Summary

node	Retrieve the leaf node of a tree with one search result.
nodes	Retrieve all leaf nodes in the search results.
path	Retrieve the path to the leaf node of a tree with one search result.
paths	Retrieve the paths to all leaf nodes in the search results.

Methods Summary

format([max_rows, max_cols, show_values])	Change formatting parameters of the rendered tree.
replace(value)	Assign a new value in place of all leaf nodes in the search results.
search([key, type, value, filter])	Further narrow the search.

Attributes Documentation

node

Retrieve the leaf node of a tree with one search result.

Returns

object

the single node of the search result

nodes

Retrieve all leaf nodes in the search results.

Returns

list of object

every node in the search results (breadth-first order)

path

Retrieve the path to the leaf node of a tree with one search result.

Returns

str

the path to the searched node

paths

Retrieve the paths to all leaf nodes in the search results.

Returns

list of str

the path to every node in the search results

Methods Documentation

format(*max_rows=NotSet*, *max_cols=NotSet*, *show_values=NotSet*)

Change formatting parameters of the rendered tree.

Parameters

max_rows

[int, tuple, None, or NotSet, optional] Maximum number of lines to print. Nodes that cannot be displayed will be elided with a message. If int, constrain total number of displayed lines. If tuple, constrain lines per node at the depth corresponding to the tuple index. If None, display all lines. If NotSet, retain existing value.

max_cols

[int, None or NotSet, optional] Maximum length of line to print. Nodes that cannot be fully displayed will be truncated with a message. If int, constrain length of displayed lines. If None, line length is unconstrained. If NotSet, retain existing value.

show_values

[bool or NotSet, optional] Set to False to disable display of primitive values in the rendered tree. Set to NotSet to retain existign value.

Returns

AsdfSearchResult

the reformatted search result

`replace`(*value*)

Assign a new value in place of all leaf nodes in the search results.

Parameters

value

[object]

`search`(*key=NotSet*, *type=NotSet*, *value=NotSet*, *filter=None*)

Further narrow the search.

Parameters

key

[NotSet, str, or any other object] Search query that selects nodes by dict key or list index. If NotSet, the node key is unconstrained. If str, the input is searched among keys/indexes as a regular expression pattern. If any other object, node's key or index must equal the queried key.

type

[NotSet, str, or builtins.type] Search query that selects nodes by type. If NotSet, the node type is unconstrained. If str, the input is searched among (fully qualified) node type names as a regular expression pattern. If builtins.type, the node must be an instance of the input.

value

[NotSet, str, or any other object] Search query that selects nodes by value. If NotSet, the node value is unconstrained. If str, the input is searched among values as a regular expression pattern. If any other object, node's value must equal the queried value.

filter

[callable] Callable that filters nodes by arbitrary criteria. The callable accepts one or two arguments:

- the node
- the node's list index or dict key (optional)

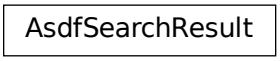
and returns True to retain the node, or False to remove it from the search results.

Returns

AsdfSearchResult

the subsequent search result

15.2.2 Class Inheritance Diagram



15.3 asdf.config Module

Methods for getting and setting asdf global configuration options.

15.3.1 Functions

get_config()	Get the current config, which may have been altered by one or more surrounding calls to asdf.config_context.
config_context()	Context manager that temporarily overrides asdf configuration.

get_config

asdf.config.**get_config**()

Get the current config, which may have been altered by one or more surrounding calls to asdf.config_context.

Returns

asdf.config.AsdfConfig

config_context

asdf.config.**config_context**()

Context manager that temporarily overrides asdf configuration. The context yields an asdf.config.AsdfConfig instance that can be modified without affecting code outside of the context.

15.3.2 Classes

AsdfConfig()	Container for ASDF configuration options.

AsdfConfig

class asdf.config.**AsdfConfig**

Bases: object

Container for ASDF configuration options. Users are not intended to construct this object directly; instead, use the asdf.get_config and asdf.config_context module methods.

Attributes Summary

array_inline_threshold	Get the threshold below which arrays are automatically written as inline YAML literals instead of binary blocks.
default_version	Get the default ASDF Standard version used for new files.
extensions	Get the list of registered extensions.
io_block_size	Get the block size used when reading and writing files.
legacy_fill_schema_defaults	Get the configuration that controls filling defaults from schemas for older ASDF Standard versions.
resource_manager	Get the ResourceManager instance.
resource_mappings	Get the list of registered resource mapping instances.
validate_on_read	Get configuration that controls schema validation of ASDF files on read.

Methods Summary

add_extension(extension)	Register a new extension.
add_resource_mapping(mapping)	Register a new resource mapping.
remove_extension([extension, package])	Remove a registered extension.
remove_resource_mapping([mapping, package])	Remove a registered resource mapping.
reset_extensions()	Reset extensions to the default list registered via entry points.
reset_resources()	Reset registered resource mappings to the default list provided as entry points.

Attributes Documentation

array_inline_threshold

Get the threshold below which arrays are automatically written as inline YAML literals instead of binary blocks. This number is compared to number of elements in the array.

Returns

int or None

Integer threshold, or None to disable automatic selection of the array storage type.

default_version

Get the default ASDF Standard version used for new files.

Returns

str

extensions

Get the list of registered extensions.

Returns

list of asdf.extension.ExtensionProxy

io_block_size

Get the block size used when reading and writing files.

Returns

int

Block size, or -1 to use the filesystem's preferred block size.

legacy_fill_schema_defaults

Get the configuration that controls filling defaults from schemas for older ASDF Standard versions. If `True`, missing default values will be filled from the schema when reading files from ASDF Standard <= 1.5.0. Later versions of the standard do not support removing or filling schema defaults.

Returns

bool

`resource_manager`

Get the `ResourceManager` instance. Includes resources from registered resource mappings and any mappings added at runtime.

Returns

asdf.resource.ResourceManager

`resource_mappings`

Get the list of registered resource mapping instances. Unless overridden by user configuration, this list contains every mapping registered with an entry point.

Returns

list of asdf.resource.ResourceMappingProxy

`validate_on_read`

Get configuration that controls schema validation of ASDF files on read.

Returns

bool

Methods Documentation

`add_extension`(*extension*)

Register a new extension. The new extension will take precedence over all previously registered extensions.

Parameters

extension

[asdf.extension.AsdfExtension or asdf.extension.Extension]

add_resource_mapping(*mapping*)

Register a new resource mapping. The new mapping will take precedence over all previously registered mappings.

Parameters

mapping

[collections.abc.Mapping] Map of `str` resource URI to `bytes` content

remove_extension(*extension=None*, *, *package=None*)

Remove a registered extension.

Parameters

extension

[asdf.extension.AsdfExtension or asdf.extension.Extension or str, optional] An extension instance or URI pattern to remove.

package

[str, optional] Remove only extensions provided by this package. If the `extension` argument is omitted, then all extensions from this package will be removed.

remove_resource_mapping(*mapping=None*, *, *package=None*)

Remove a registered resource mapping.

Parameters

mapping

[collections.abc.Mapping, optional] Mapping to remove.

package

[str, optional] Remove only extensions provided by this package. If the `mapping` argument is omitted, then all mappings from this package will be removed.

reset_extensions()

Reset extensions to the default list registered via entry points.

reset_resources()

Reset registered resource mappings to the default list provided as entry points.

15.3.3 Class Inheritance Diagram

CHAPTER

SIXTEEN

DEVELOPER API

The classes and functions documented here will be of use to developers who wish to create their own custom ASDF types and extensions.

16.1 asdf.types Module

16.1.1 Functions

format_tag(organization, standard, version, ...)	Format a YAML tag.

format_tag

asdf.types.**format_tag**(*organization*, *standard*, *version*, *tag_name*)

Format a YAML tag.

16.1.2 Classes

CustomType()	Base class for all user-defined types.

CustomType

class asdf.types.**CustomType**

Bases: ExtensionType

Base class for all user-defined types.

Attributes Summary

handle_dynamic_subclasses	bool: Indicates whether dynamically generated subclasses can be serialized
has_required_modules	bool: Indicates whether modules specified by requires are available.
name	str or list: The name of the type.
organization	str: The organization responsible for the type.
requires	list: Python packages that are required to instantiate the object.
standard	str: The standard the type is defined in.
supported_versions	set: Versions that explicitly compatible with this extension class.
types	list: List of types that this extension class can convert to/from YAML.
validators	dict: Mapping JSON Schema keywords to validation functions for jsonschema.
version	str, tuple, AsdfVersion, or AsdfSpec: The version of the type.
yaml_tag	str: The YAML tag to use for the type.

Attributes Documentation

handle_dynamic_subclasses = False

bool: Indicates whether dynamically generated subclasses can be serialized

Flag indicating whether this type is capable of serializing subclasses of any of the types listed in types that are generated dynamically.

has_required_modules = True

bool: Indicates whether modules specified by requires are available.

NOTE: This value is automatically generated. Do not set it in subclasses as it will be overwritten.

name = None

str or list: The name of the type.

organization = 'stsci.edu'

str: The organization responsible for the type.

requires = []

list: Python packages that are required to instantiate the object.

standard = 'asdf'

str: The standard the type is defined in.

supported_versions = []

set: Versions that explicitly compatible with this extension class.

If provided, indicates explicit compatibility with the given set of versions. Other versions of the same schema that are not included in this set will not be converted to custom types with this class.

types = []

list: List of types that this extension class can convert to/from YAML.

Custom Python types that, when found in the tree, will be converted into basic types for YAML output. Can be either strings referring to the types or the types themselves.

validators = {}

dict: Mapping JSON Schema keywords to validation functions for jsonschema.

Useful if the type defines extra types of validation that can be performed.

version = AsdfVersion('1.0.0')

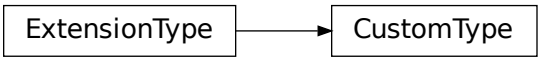
str, tuple, AsdfVersion, or AsdfSpec: The version of the type.

yaml_tag = None

str: The YAML tag to use for the type.

If not provided, it will be automatically generated from name, organization, standard and version.

16.1.3 Class Inheritance Diagram



16.2 asdf.extension Package

Support for plugins that extend asdf to serialize additional custom types.

16.2.1 Functions

get_cached_extension_manager(extensions)	Get a previously created ExtensionManager for the specified extensions, or create and cache one if necessary.
get_default_resolver()	Get the resolver that includes mappings from all installed extensions.
get_cached_asdf_extension_list(extensions)	Get a previously created AsdfExtensionList for the specified extensions, or create and cache one if necessary.

get_cached_extension_manager

asdf.extension.**get_cached_extension_manager**(*extensions*)

Get a previously created ExtensionManager for the specified extensions, or create and cache one if necessary. Building the manager is expensive, so it helps performance to reuse it when possible.

Parameters

extensions

[list of asdf.extension.AsdfExtension or asdf.extension.Extension]

Returns

asdf.extension.ExtensionManager

get_default_resolver

asdf.extension.**get_default_resolver**()

Get the resolver that includes mappings from all installed extensions.

get_cached_asdf_extension_list

asdf.extension.**get_cached_asdf_extension_list**(*extensions*)

Get a previously created AsdfExtensionList for the specified extensions, or create and cache one if necessary. Building the type index is expensive, so it helps performance to reuse the index when possible.

Parameters

extensions

[list of asdf.extension.AsdfExtension]

Returns

asdf.extension.AsdfExtensionList

16.2.2 Classes

Extension()	Abstract base class defining an extension to ASDF.
ExtensionProxy(delegate[, package_name, ...])	Proxy that wraps an extension, provides default implementations of optional methods, and carries additional information on the package that provided the extension.
ManifestExtension(manifest, *[, ...])	Extension implementation that reads the extension URI, ASDF Standard requirement, and tag list from a manifest document.
ExtensionManager(extensions)	Wraps a list of extensions and indexes their converters by tag and by Python type.
TagDefinition(tag_uri, *[, schema_uris, ...])	Container for properties of a custom YAML tag.
Converter()	Abstract base class for plugins that convert nodes from the parsed YAML tree into custom objects, and vice versa.
ConverterProxy(delegate, extension)	Proxy that wraps a Converter and provides default implementations of optional methods.
Compressor()	Abstract base class for plugins that compress binary data.
AsdfExtension()	Abstract base class defining a (legacy) extension to ASDF.
AsdfExtensionList(extensions)	Manage a set of extensions that are in effect.
BuiltinExtension()	This is the "extension" to ASDF that includes all the built-in tags.

Extension

class asdf.extension.**Extension**

Bases: ABC

Abstract base class defining an extension to ASDF.

Implementing classes must provide the extension_uri. Other properties are optional.

Attributes Summary

asdf_standard_requirement	Get the ASDF Standard version requirement for this extension.
compressors	Get the asdf.extension.Compressor instances for compression schemes supported by this extension.
converters	Get the asdf.extension.Converter instances for tags and Python types supported by this extension.
extension_uri	Get the URI of the extension to the ASDF Standard implemented by this class.
legacy_class_names	Get the set of fully-qualified class names used by older versions of this extension.
tags	Get the YAML tags supported by this extension.
yaml_tag_handles	Get a dictionary of custom yaml TAG handles defined by the extension.

Attributes Documentation

asdf_standard_requirement

Get the ASDF Standard version requirement for this extension.

Returns

str or None

If str, PEP 440 version specifier. If None, support all versions.

compressors

Get the asdf.extension.Compressor instances for compression schemes supported by this extension.

Returns

iterable of asdf.extension.Compressor

converters

Get the asdf.extension.Converter instances for tags and Python types supported by this extension.

Returns

iterable of asdf.extension.Converter

`extension_uri`

Get the URI of the extension to the ASDF Standard implemented by this class. Note that this may not uniquely identify the class itself.

Returns

str

`legacy_class_names`

Get the set of fully-qualified class names used by older versions of this extension. This allows a new-style implementation of an extension to prevent warnings when a legacy extension is missing.

Returns

iterable of str

`tags`

Get the YAML tags supported by this extension.

Returns

iterable of str or asdf.extension.TagDefinition

`yaml_tag_handles`

Get a dictionary of custom yaml TAG handles defined by the extension.

The dictionary key indicates the TAG handles to be placed in the YAML header, the value defines the string for tag replacement. See https://yaml.org/spec/1.1/#tag%20shorthand/

Example: `{"!foo!": "tag:nowhere.org:custom/"}`

Returns

dict

ExtensionProxy

class asdf.extension.**ExtensionProxy**(*delegate*, *package_name=None*, *package_version=None*)

Bases: Extension, AsdfExtension

Proxy that wraps an extension, provides default implementations of optional methods, and carries additional information on the package that provided the extension.

Attributes Summary

asdf_standard_requirement	Get the extension's ASDF Standard requirement.
class_name	Get the fully qualified class name of the extension.
compressors	Get the extension's compressors.
converters	Get the extension's converters.
delegate	Get the wrapped extension instance.
extension_uri	Get the URI of the extension to the ASDF Standard implemented by this class.
legacy	Get the extension's legacy flag.
legacy_class_names	Get the set of fully-qualified class names used by older versions of this extension.
package_name	Get the name of the Python package that provided this extension.
package_version	Get the version of the Python package that provided the extension
tag_mapping	Get the legacy extension's tag-to-schema-URI mapping.
tags	Get the YAML tags supported by this extension.
types	Get the legacy extension's ExtensionType subclasses.
url_mapping	Get the legacy extension's schema-URI-to-URL mapping.
yaml_tag_handles	Get a dictionary of custom yaml TAG handles defined by the extension.

Methods Summary

maybe_wrap(delegate)	

Attributes Documentation

asdf_standard_requirement

Get the extension's ASDF Standard requirement.

Returns

packaging.specifiers.SpecifierSet

`class_name`

Get the fully qualified class name of the extension.

Returns

str

`compressors`

Get the extension's compressors.

Returns

list of asdf.extension.Compressor

`converters`

Get the extension's converters.

Returns

list of asdf.extension.Converter

`delegate`

Get the wrapped extension instance.

Returns

asdf.extension.Extension or asdf.extension.AsdfExtension

`extension_uri`

Get the URI of the extension to the ASDF Standard implemented by this class. Note that this may not uniquely identify the class itself.

Returns

str or None

`legacy`

Get the extension's legacy flag. Subclasses of `asdf.extension.AsdfExtension` are marked `True`.

Returns

bool

`legacy_class_names`

Get the set of fully-qualified class names used by older versions of this extension. This allows a new-style implementation of an extension to prevent warnings when a legacy extension is missing.

Returns

set of str

`package_name`

Get the name of the Python package that provided this extension.

Returns

str or None

`None` if the extension was added at runtime.

`package_version`

Get the version of the Python package that provided the extension

Returns

str or None

`None` if the extension was added at runtime.

`tag_mapping`

Get the legacy extension's tag-to-schema-URI mapping.

Returns

iterable of tuple or callable

`tags`

Get the YAML tags supported by this extension.

Returns

list of asdf.extension.TagDefinition

`types`

Get the legacy extension's ExtensionType subclasses.

Returns

iterable of asdf.type.ExtensionType

`url_mapping`

Get the legacy extension's schema-URI-to-URL mapping.

Returns

iterable of tuple or callable

`yaml_tag_handles`

Get a dictionary of custom yaml TAG handles defined by the extension.

The dictionary key indicates the TAG handles to be placed in the YAML header, the value defines the string for tag replacement. See https://yaml.org/spec/1.1/#tag%20shorthand/

Example: `{"!foo!": "tag:nowhere.org:custom/"}`

Returns

dict

Methods Documentation

classmethod maybe_wrap(*delegate*)

ManifestExtension

class asdf.extension.**ManifestExtension**(*manifest*, ***, *legacy_class_names=None*, *converters=None*, *compressors=None*)

Bases: Extension

Extension implementation that reads the extension URI, ASDF Standard requirement, and tag list from a manifest document.

Parameters

manifest

[dict] Parsed manifest.

converters

[iterable of asdf.extension.Converter, optional] Converter instances for the tags and Python types supported by this extension.

compressors

[iterable of asdf.extension.Compressor, optional] Compressor instances to support additional binary block compression options.

legacy_class_names

[iterable of str, optional] Fully-qualified class names used by older versions of this extension.

Attributes Summary

asdf_standard_requirement	Get the ASDF Standard version requirement for this extension.
compressors	Get the asdf.extension.Compressor instances for compression schemes supported by this extension.
converters	Get the asdf.extension.Converter instances for tags and Python types supported by this extension.
extension_uri	Get the URI of the extension to the ASDF Standard implemented by this class.
legacy_class_names	Get the set of fully-qualified class names used by older versions of this extension.
tags	Get the YAML tags supported by this extension.

Methods Summary

from_uri(manifest_uri, **kwargs)	Construct the extension using the manifest with the specified URI.

Attributes Documentation

asdf_standard_requirement

compressors

converters

extension_uri

legacy_class_names

tags

Methods Documentation

classmethod from_uri(*manifest_uri*, ***kwargs*)

Construct the extension using the manifest with the specified URI. The manifest document must be registered with ASDF's resource manager.

Parameters

manifest_uri

[str] Manifest URI.

See the class docstring for details on keyword parameters.

ExtensionManager

class asdf.extension.**ExtensionManager**(*extensions*)

Bases: object

Wraps a list of extensions and indexes their converters by tag and by Python type.

Parameters

extensions

[iterable of asdf.extension.Extension] List of enabled extensions to manage. Extensions placed earlier in the list take precedence.

Attributes Summary

extensions	Get the list of extensions.

Methods Summary

get_converter_for_tag(tag)	Get the converter for the specified tag.
get_converter_for_type(typ)	Get the converter for the specified Python type.
get_tag_definition(tag)	Get the tag definition for the specified tag.
handles_tag(tag)	Return True if the specified tag is handled by a converter.
handles_tag_definition(tag)	Return True if the specified tag has a definition.
handles_type(typ)	Returns True if the specified Python type is handled by a converter.

Attributes Documentation

extensions

Get the list of extensions.

Returns

list of asdf.extension.ExtensionProxy

Methods Documentation

get_converter_for_tag(*tag*)

Get the converter for the specified tag.

Parameters

tag

[str] Tag URI.

Returns

asdf.extension.Converter

Raises

KeyError

Unrecognized tag URI.

get_converter_for_type(*typ*)

Get the converter for the specified Python type.

Parameters

typ

[type]

Returns

asdf.extension.Converter

Raises

KeyError

Unrecognized type.

get_tag_definition(*tag*)

Get the tag definition for the specified tag.

Parameters

tag

[str] Tag URI.

Returns

asdf.extension.TagDefinition

Raises

KeyError

Unrecognized tag URI.

`handles_tag`(*tag*)

Return `True` if the specified tag is handled by a converter.

Parameters

tag

[str] Tag URI.

Returns

bool

`handles_tag_definition`(*tag*)

Return `True` if the specified tag has a definition.

Parameters

tag

[str] Tag URI.

Returns

bool

handles_type(*typ*)

Returns `True` if the specified Python type is handled by a converter.

Parameters

typ

[type]

Returns

bool

TagDefinition

class asdf.extension.**TagDefinition**(*tag_uri*, *, *schema_uris=None*, *title=None*, *description=None*)

Bases: `object`

Container for properties of a custom YAML tag.

Parameters

tag_uri

[str] Tag URI.

schema_uri

[str, optional] URI of the schema that should be used to validate objects with this tag.

title

[str, optional] Short description of the tag.

description

[str, optional] Long description of the tag.

Attributes Summary

description	Get the long description of the tag.
schema_uri	DEPRECATED
schema_uris	Get the URIs of the schemas that should be used to validate objects with this tag.
tag_uri	Get the tag URI.
title	Get the short description of the tag.

Attributes Documentation

description

Get the long description of the tag.

Returns

str or None

schema_uri

DEPRECATED

Get the URI of the schema that should be used to validate objects with this tag.

Returns

str or None

schema_uris

Get the URIs of the schemas that should be used to validate objects with this tag.

Returns

list

tag_uri

Get the tag URI.

Returns

str

title

Get the short description of the tag.

Returns

str or None

Converter

class asdf.extension.**Converter**

Bases: ABC

Abstract base class for plugins that convert nodes from the parsed YAML tree into custom objects, and vice versa.

Implementing classes must provide the `tags` and `types` properties and `to_yaml_tree` and `from_yaml_tree` methods. The `select_tag` method is optional.

Attributes Summary

`tags`	Get the YAML tags that this converter is capable of handling.
`types`	Get the Python types that this converter is capable of handling.

Methods Summary

`from_yaml_tree`(node, tag, ctx)	Convert a YAML node into an instance of a custom type.
`select_tag`(obj, tags, ctx)	Select the tag to use when converting an object to YAML.
`to_yaml_tree`(obj, tag, ctx)	Convert an object into a node suitable for YAML serialization.

Attributes Documentation

tags

Get the YAML tags that this converter is capable of handling. URI patterns are permitted, see asdf.util.uri_match for details.

Returns

iterable of str

Tag URIs or URI patterns.

types

Get the Python types that this converter is capable of handling.

Returns

iterable of str or type

If str, the fully qualified class name of the type.

Methods Documentation

abstract from_yaml_tree(*node*, *tag*, *ctx*)

Convert a YAML node into an instance of a custom type.

For container types received by this method (dict or list), the children of the container will have already been converted by prior calls to from_yaml_tree implementations.

Note on circular references: trees that reference themselves among their descendants must be handled with care. Most implementations need not concern themselves with this case, but if the custom type supports circular references, then the implementation of this method will need to return a generator. Consult the documentation for more details.

Parameters

node

[dict or list or str] The YAML node to convert.

tag

[str] The YAML tag of the object being converted.

ctx

[asdf.asdf.SerializationContext] The context of the current deserialization request.

Returns

object

An instance of one of the types listed in the `types` property, or a generator that yields such an instance.

`select_tag`(*obj*, *tags*, *ctx*)

Select the tag to use when converting an object to YAML. Typically only one tag will be active in a given context, but converters that map one type to many tags must provide logic to choose the appropriate tag.

Parameters

obj

[object] Instance of the custom type being converted. Guaranteed to be an instance of one of the types listed in the `types` property.

tags

[list of str] List of active tags to choose from. Guaranteed to match one of the tag patterns listed in the 'tags' property.

ctx

[asdf.asdf.SerializationContext] Context of the current serialization request.

Returns

str

The selected tag. Should be one of the tags passed to this method in the `tags` parameter.

`abstract to_yaml_tree`(*obj*, *tag*, *ctx*)

Convert an object into a node suitable for YAML serialization. This method is not responsible for writing actual YAML; rather, it converts an instance of a custom type to a built-in Python object type (such as dict, list, str, or number), which can then be automatically serialized to YAML as needed.

For container types returned by this method (dict or list), the children of the container need not themselves be converted. Any list elements or dict values will be converted by subsequent calls to to_yaml_tree implementations.

The returned node must be an instance of `dict`, `list`, or `str`. Children may be any type supported by an available Converter.

Parameters

obj

[object] Instance of a custom type to be serialized. Guaranteed to be an instance of one of the types listed in the `types` property.

tag

[str] The tag identifying the YAML type that `obj` should be converted into. Selected by a call to this converter's select_tag method.

ctx

[asdf.asdf.SerializationContext] The context of the current serialization request.

Returns

dict or list or str

The YAML node representation of the object.

ConverterProxy

class asdf.extension.**ConverterProxy**(*delegate*, *extension*)

Bases: `Converter`

Proxy that wraps a `Converter` and provides default implementations of optional methods.

Attributes Summary

`class_name`	Get the fully qualified class name of this converter.
`delegate`	Get the wrapped converter instance.
`extension`	Get the extension that provided this converter.
`package_name`	Get the name of the Python package of this converter's extension.
`package_version`	Get the version of the Python package of this converter's extension.
`tags`	Get the list of tag URIs that this converter is capable of handling.
`types`	Get the Python types that this converter is capable of handling.

Methods Summary

`from_yaml_tree`(node, tag, ctx)	Convert a YAML node into an instance of a custom type.
`select_tag`(obj, ctx)	Select the tag to use when converting an object to YAML.
`to_yaml_tree`(obj, tag, ctx)	Convert an object into a node suitable for YAML serialization.

Attributes Documentation

class_name

Get the fully qualified class name of this converter.

Returns

str

delegate

Get the wrapped converter instance.

Returns

asdf.extension.Converter

extension

Get the extension that provided this converter.

Returns

asdf.extension.ExtensionProxy

package_name

Get the name of the Python package of this converter's extension. This may not be the same package that implements the converter's class.

Returns

str or None

Package name, or `None` if the extension was added at runtime.

package_version

Get the version of the Python package of this converter's extension. This may not be the same package that implements the converter's class.

Returns

str or None

Package version, or None if the extension was added at runtime.

tags

Get the list of tag URIs that this converter is capable of handling.

Returns

list of str

types

Get the Python types that this converter is capable of handling.

Returns

list of type or str

Methods Documentation

from_yaml_tree(*node*, *tag*, *ctx*)

Convert a YAML node into an instance of a custom type.

Parameters

tree

[dict or list or str] The YAML node to convert.

tag

[str] The YAML tag of the object being converted.

ctx

[asdf.asdf.SerializationContext] Serialization parameters.

Returns

object

`select_tag`(*obj*, *ctx*)

Select the tag to use when converting an object to YAML.

Parameters

obj

[object] Instance of the custom type being converted.

ctx

[asdf.asdf.SerializationContext] Serialization parameters.

Returns

str

Selected tag.

`to_yaml_tree`(*obj*, *tag*, *ctx*)

Convert an object into a node suitable for YAML serialization.

Parameters

obj

[object] Instance of a custom type to be serialized.

tag

[str] The tag identifying the YAML type that `obj` should be converted into.

ctx

[asdf.asdf.SerializationContext] Serialization parameters.

Returns

object

The YAML node representation of the object.

Compressor

class asdf.extension.**Compressor**

Bases: ABC

Abstract base class for plugins that compress binary data.

Implementing classes must provide the `labels` property, and at least one of the `compress()` and `decompress()` methods. May also provide a constructor.

Attributes Summary

`label`	Get the 4-byte label identifying this compression

Methods Summary

`compress`(data, **kwargs)	Compress `data`, yielding the results.
`decompress`(data, out, **kwargs)	Decompress `data`, writing the result into `out`.

Attributes Documentation

label

Get the 4-byte label identifying this compression

Returns

label

[bytes] The compression label

Methods Documentation

compress(*data*, ***kwargs*)

Compress `data`, yielding the results. The yield may be block-by-block, or all at once.

Parameters

data

[memoryview] The data to compress. Must be contiguous and 1D, with the underlying `itemsize` preserved.

****kwargs**

Keyword arguments to be passed to the underlying compression function

Yields

compressed

[bytes-like] A block of compressed data

`decompress`(*data*, *out*, ***kwargs*)

Decompress `data`, writing the result into `out`.

Parameters

data

[Iterable of bytes-like] An Iterable of bytes-like objects containing chunks of compressed data.

out

[read-write bytes-like] A contiguous, 1D output array, of equal or greater length than the decompressed data.

****kwargs**

Keyword arguments to be passed to the underlying decompression function

Returns

nbytes

[int] The number of bytes written to `out`

AsdfExtension

`class` asdf.extension.`AsdfExtension`

Bases: `object`

Abstract base class defining a (legacy) extension to ASDF. New code should use `asdf.extension.Extension` instead.

Attributes Summary

tag_mapping	A list of 2-tuples or callables mapping YAML tag prefixes to JSON Schema URL prefixes.
types	A list of asdf.CustomType subclasses that describe how to store custom objects to and from ASDF.
url_mapping	Schema content can be provided using the resource Mapping API.

Attributes Documentation

tag_mapping

A list of 2-tuples or callables mapping YAML tag prefixes to JSON Schema URL prefixes.

For each entry:

- If a 2-tuple, the first part of the tuple is a YAML tag prefix to match. The second part is a string, where case the following are available as Python formatting tokens:
  - {tag}: the complete YAML tag.
  - {tag_suffix}: the part of the YAML tag after the matched prefix.
  - {tag_prefix}: the matched YAML tag prefix.
- If a callable, it is passed the entire YAML tag must return the entire JSON schema URL if it matches, otherwise, return None.

Note that while JSON Schema URLs uniquely define a JSON Schema, they do not have to actually exist on an HTTP server and be fetchable (much like XML namespaces).

For example, to match all YAML tags with the tag:nowhere.org:custom` prefix to the ``http://nowhere.org/schemas/custom/ URL prefix:

```
return [('tag:nowhere.org:custom/',
         'http://nowhere.org/schemas/custom/{tag_suffix}')]
```

types

A list of asdf.CustomType subclasses that describe how to store custom objects to and from ASDF.

url_mapping

Schema content can be provided using the resource Mapping API.

A list of 2-tuples or callables mapping JSON Schema URLs to other URLs. This is useful if the JSON Schemas are not actually fetchable at their corresponding URLs but are on the local filesystem, or, to save bandwidth, we have a copy of fetchable schemas on the local filesystem. If neither is desirable, it may simply be the empty list.

For each entry:

- If a 2-tuple, the first part is a URL prefix to match. The second part is a string, where the following are available as Python formatting tokens:
  - {url}: The entire JSON schema URL
  - {url_prefix}: The matched URL prefix

  - `{url_suffix}`: The part of the URL after the prefix.

- If a callable, it is passed the entire JSON Schema URL and must return a resolvable URL pointing to the schema content. If it doesn't match, should return `None`.

For example, to map a remote HTTP URL prefix to files installed alongside as data alongside Python module:

```
return [('http://nowhere.org/schemas/custom/1.0.0/',
        asdf.util.filepath_to_url(
            os.path.join(SCHEMA_PATH, 'stsci.edu')) +
        '/{url_suffix}.yaml'
       )]
```

AsdfExtensionList

class asdf.extension.**AsdfExtensionList**(*extensions*)

Bases: `object`

Manage a set of extensions that are in effect.

Attributes Summary

`extensions`	
`resolver`	
`tag_mapping`	
`tag_to_schema_resolver`	Deprecated.
`type_index`	
`url_mapping`	
`validators`	

Attributes Documentation

extensions

resolver

tag_mapping

tag_to_schema_resolver

Deprecated. Use tag_mapping instead

type_index

url_mapping

validators

BuiltinExtension

class asdf.extension.**BuiltinExtension**

Bases: object

This is the "extension" to ASDF that includes all the built-in tags. Even though it's not really an extension and it's always available, it's built in the same way as an extension.

Attributes Summary

tag_mapping
types
url_mapping

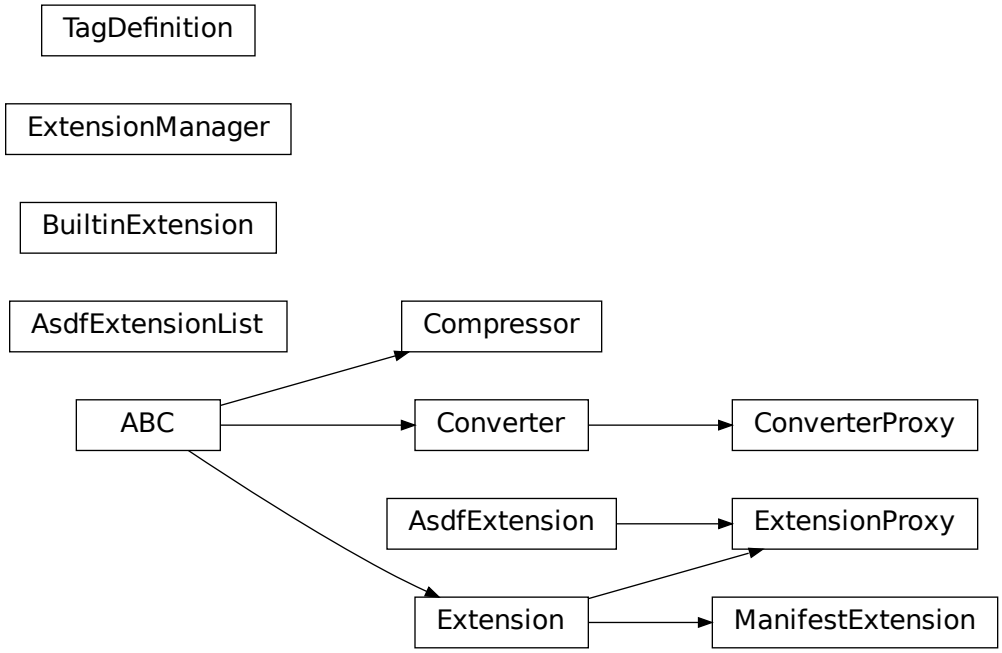
Attributes Documentation

tag_mapping

types

url_mapping

16.2.3 Class Inheritance Diagram



16.3 asdf.resource Module

Support for plugins that provide access to resources such as schemas.

16.3.1 Functions

`get_json_schema_resource_mappings`()

get_json_schema_resource_mappings

asdf.resource.**get_json_schema_resource_mappings**()

16.3.2 Classes

ResourceMappingProxy(delegate[, ...])	Wrapper around a resource mapping that carries additional information on the package that provided the mapping.
DirectoryResourceMapping(root, uri_prefix[, ...])	A resource mapping that reads resource content from a directory or directory tree.
ResourceManager(resource_mappings)	Wraps multiple resource mappings into a single interface with some friendlier error handling.
JsonschemaResourceMapping()	Resource mapping that fetches metaschemas from the jsonschema package.

ResourceMappingProxy

class asdf.resource.**ResourceMappingProxy**(*delegate*, *package_name=None*, *package_version=None*)

Bases: Mapping

Wrapper around a resource mapping that carries additional information on the package that provided the mapping.

Attributes Summary

class_name	"
delegate	Get the wrapped mapping instance.
package_name	Get the name of the Python package that provided this mapping.
package_version	Get the version of the Python package that provided the mapping.

Methods Summary

maybe_wrap(delegate)

Attributes Documentation

class_name

" Get the fully qualified class name of the mapping.

Returns

str

delegate

Get the wrapped mapping instance.

Returns

collections.abc.Mapping

package_name

Get the name of the Python package that provided this mapping.

Returns

str or None

None if the mapping was added at runtime.

package_version

Get the version of the Python package that provided the mapping.

Returns

str or None

None if the mapping was added at runtime.

Methods Documentation

classmethod maybe_wrap(*delegate*)

DirectoryResourceMapping

class asdf.resource.**DirectoryResourceMapping**(*root*, *uri_prefix*, *recursive=False*, *filename_pattern='*.yaml'*, *stem_filename=True*)

Bases: DirectoryResourceMapping

A resource mapping that reads resource content from a directory or directory tree.

See DirectoryResourceMapping for details.

ResourceManager

class asdf.resource.**ResourceManager**(*resource_mappings*)

Bases: Mapping

Wraps multiple resource mappings into a single interface with some friendlier error handling.

Parameters

resource_mappings

[iterable of collections.abc.Mapping] Underlying resource mappings. In the case of a duplicate URI, the first mapping takes precedence.

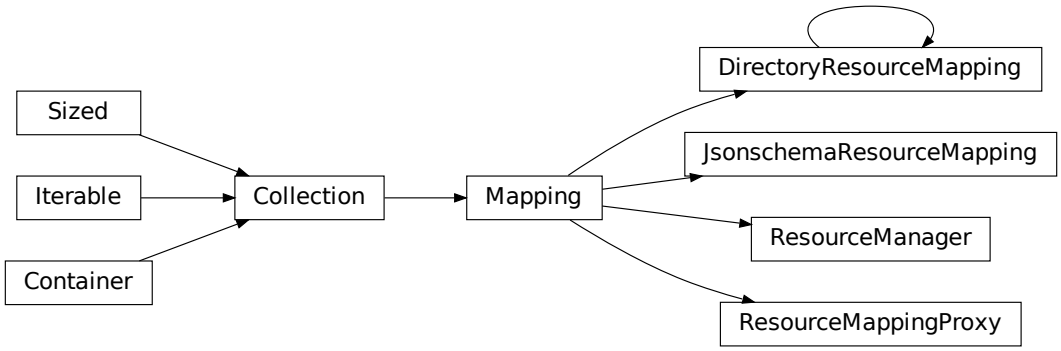
JsonschemaResourceMapping

class asdf.resource.**JsonschemaResourceMapping**

Bases: Mapping

Resource mapping that fetches metaschemas from the jsonschema package.

16.3.3 Class Inheritance Diagram



16.4 asdf.yamlutil Module

16.4.1 Functions

custom_tree_to_tagged_tree(tree, ctx[, ...])	Convert a tree, possibly containing custom data types that aren't directly representable in YAML, to a tree of basic data types, annotated with tags.
tagged_tree_to_custom_tree(tree, ctx[, ...])	Convert a tree containing only basic data types, annotated with tags, to a tree containing custom data types.

custom_tree_to_tagged_tree

asdf.yamlutil.**custom_tree_to_tagged_tree**(*tree*, *ctx*, *_serialization_context=None*)

Convert a tree, possibly containing custom data types that aren't directly representable in YAML, to a tree of basic data types, annotated with tags.

tagged_tree_to_custom_tree

asdf.yamlutil.**tagged_tree_to_custom_tree**(*tree*, *ctx*, *force_raw_types=False*, *_serialization_context=None*)

Convert a tree containing only basic data types, annotated with tags, to a tree containing custom data types.

16.5 asdf.util Module

16.5.1 Functions

human_list(l[, separator])	Formats a list for human readability.
get_array_base(arr)	For a given Numpy array, finds the base array that "owns" the actual data.
get_base_uri(uri)	For a given URI, return the part without any fragment.
filepath_to_url(path)	For a given local file path, return a file:// url.
iter_subclasses(cls)	Returns all subclasses of a class.
calculate_padding(content_size, pad_blocks, ...)	Calculates the amount of extra space to add to a block given the user's request for the amount of extra space.
resolve_name(name)	Resolve a name like `module.object` to an object and return it.
is_primitive(value)	Determine if a value is an instance of a "primitive" type.
uri_match(pattern, uri)	Determine if a URI matches a URI pattern with possible wildcards.
get_class_name(obj[, instance])	Given a class or instance of a class, returns a string representing the fully specified path of the class.

human_list

asdf.util.**human_list**(*l*, *separator='and'*)

Formats a list for human readability.

Parameters

l

[sequence] A sequence of strings

separator

[string, optional] The word to use between the last two entries. Default: "and".

Returns

formatted_list

[string]

Examples

```
>>> human_list(["vanilla", "strawberry", "chocolate"], "or")
'vanilla, strawberry or chocolate'
```

get_array_base

asdf.util.**get_array_base**(*arr*)

For a given Numpy array, finds the base array that "owns" the actual data.

get_base_uri

asdf.util.**get_base_uri**(*uri*)

For a given URI, return the part without any fragment.

filepath_to_url

asdf.util.**filepath_to_url**(*path*)

For a given local file path, return a file:// url.

iter_subclasses

asdf.util.**iter_subclasses**(*cls*)

Returns all subclasses of a class.

calculate_padding

asdf.util.**calculate_padding**(*content_size*, *pad_blocks*, *block_size*)

Calculates the amount of extra space to add to a block given the user's request for the amount of extra space. Care is given so that the total of size of the block with padding is evenly divisible by block size.

Parameters

content_size

[int] The size of the actual content

pad_blocks

[float or bool] If `False`, add no padding (always return 0). If `True`, add a default amount of padding of 10% If a float, it is a factor to multiple content_size by to get the new total size.

block_size

[int] The filesystem block size to use.

Returns

nbytes

[int] The number of extra bytes to add for padding.

resolve_name

asdf.util.**resolve_name**(*name*)

Resolve a name like `module.object` to an object and return it.

This ends up working like `from module import object` but is easier to deal with than the `__import__` builtin and supports digging into submodules.

Parameters

name

[str] A dotted path to a Python object–that is, the name of a function, class, or other object in a module with the full path to that module, including parent modules, separated by dots. Also known as the fully qualified name of the object.

Raises

ImportError

If the module or named object is not found.

Examples

```
>>> resolve_name('asdf.util.resolve_name')
<function resolve_name at 0x...>
```

is_primitive

asdf.util.**is_primitive**(*value*)

Determine if a value is an instance of a "primitive" type.

Parameters

value

[object] the value to test

Returns

bool

True if the value is primitive, False otherwise

uri_match

asdf.util.**uri_match**(*pattern*, *uri*)

Determine if a URI matches a URI pattern with possible wildcards. The two recognized wildcards:

"*": match any character except /

"**": match any character

Parameters

pattern

[str] URI pattern.

uri

[str] URI to check against the pattern.

Returns

bool

`True` if URI matches the pattern.

get_class_name

asdf.util.**get_class_name**(*obj*, *instance=True*)

Given a class or instance of a class, returns a string representing the fully specified path of the class.

Parameters

obj

[object] An instance of any object

instance: bool

Indicates whether given object is an instance of the class to be named

16.6 asdf.versioning Module

This module deals with things that change between different versions of the ASDF spec.

16.6.1 Functions

split_tag_version(tag)	Split a tag into its base and version.
join_tag_version(name, version)	Join the root and version of a tag back together.

split_tag_version

asdf.versioning.**split_tag_version**(*tag*)

Split a tag into its base and version.

join_tag_version

asdf.versioning.**join_tag_version**(*name*, *version*)

Join the root and version of a tag back together.

16.6.2 Classes

AsdfVersion(version)	This class adds features to the existing Version class from the semantic_version module.
AsdfSpec(*args, **kwargs)	

AsdfVersion

class asdf.versioning.**AsdfVersion**(*version*)

Bases: AsdfVersionMixin, Version

This class adds features to the existing Version class from the semantic_version module. Namely, it allows Version objects to be constructed from tuples and lists as well as strings, and it allows Version objects to be compared with tuples, lists, and strings, instead of just other Version objects.

If any of these features are added to the Version class itself (as requested in https://github.com/rbarrois/python-semanticversion/issues/52), then this class will become obsolete.

AsdfSpec

class asdf.versioning.**AsdfSpec**(**args*, ***kwargs*)

Bases: SimpleSpec

Methods Summary

filter(versions)	Filter an iterable of versions satisfying the Spec.
match(version)	Check whether a Version satisfies the Spec.
select(versions)	Select the best compatible version among an iterable of options.

Methods Documentation

filter(*versions*)

Filter an iterable of versions satisfying the Spec.

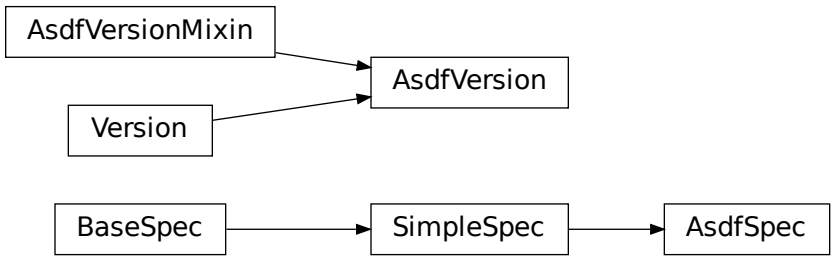
match(*version*)

Check whether a Version satisfies the Spec.

select(*versions*)

Select the best compatible version among an iterable of options.

16.6.3 Class Inheritance Diagram



16.7 asdf.tests.helpers Module

16.7.1 Functions

get_test_data_path(name[, module])	

assert_tree_match(old_tree, new_tree[, ctx, ...])	Assert that two ASDF trees match.
assert_roundtrip_tree(*args, **kwargs)	Assert that a given tree saves to ASDF and, when loaded back, the tree matches the original tree.
yaml_to_asdf(yaml_content[, yaml_headers, ...])	Given a string of YAML content, adds the extra pre- and post-amble to make it an ASDF file.
get_file_sizes(dirname)	Get the file sizes in a directory.
display_warnings(_warnings)	Return a string that displays a list of unexpected warnings

get_test_data_path

asdf.tests.helpers.**get_test_data_path**(*name*, *module=None*)

assert_tree_match

asdf.tests.helpers.**assert_tree_match**(*old_tree*, *new_tree*, *ctx=None*, *funcname='assert_equal'*, *ignore_keys=None*)

Assert that two ASDF trees match.

Parameters

old_tree

[ASDF tree]

new_tree

[ASDF tree]

ctx

[ASDF file context] Used to look up the set of types in effect.

funcname

[str or callable] The name of a method on members of old_tree and new_tree that will be used to compare custom objects. The default of `assert_equal` handles Numpy arrays.

ignore_keys

[list of str] List of keys to ignore

assert_roundtrip_tree

asdf.tests.helpers.**assert_roundtrip_tree**(**args*, ***kwargs*)

Assert that a given tree saves to ASDF and, when loaded back, the tree matches the original tree.

tree : ASDF tree

tmpdir

[str] Path to temporary directory to save file

tree_match_func

[`str` or `callable`] Passed to `assert_tree_match` and used to compare two objects in the tree.

raw_yaml_check_func

[callable, optional] Will be called with the raw YAML content as a string to perform any additional checks.

asdf_check_func

[callable, optional] Will be called with the reloaded ASDF file to perform any additional checks.

yaml_to_asdf

asdf.tests.helpers.**yaml_to_asdf**(*yaml_content*, *yaml_headers=True*, *standard_version=None*)

Given a string of YAML content, adds the extra pre- and post-amble to make it an ASDF file.

Parameters

yaml_content

[string]

yaml_headers

[bool, optional] When True (default) add the standard ASDF YAML headers.

Returns

buff

[io.BytesIO()] A file-like object containing the ASDF-like content.

get_file_sizes

asdf.tests.helpers.**get_file_sizes**(*dirname*)

Get the file sizes in a directory.

Parameters

dirname

[string] Path to a directory

Returns

sizes

[dict] Dictionary of (file, size) pairs.

display_warnings

asdf.tests.helpers.**display_warnings**(*_warnings*)

Return a string that displays a list of unexpected warnings

Parameters

_warnings

[iterable] List of warnings to be displayed

Returns

msg

[str] String containing the warning messages to be displayed

Part IV

Developer Overview

Currently a work in progress. Intended to give an overview of how the various parts of ASDF interact and which modules do what and how.

CHAPTER

SEVENTEEN

HIGH LEVEL OVERVIEW OF THE BASIC ASDF LIBRARY

This document is an attempt to make it easier to understand the design and workings of the python asdf library for those unfamiliar with it. This is expected to grow organically so at the moment it should not be considered complete or comprehensive.

Understanding the design is complicated by the fact that the library effectively inserts custom methods or classes into the objects that the pyyaml and jsonschema libraries use. Understanding what is going on thus means having some understanding of the relevant parts of the internals of both of those libraries. This overview will try to provide a small amount of context for these packages to illuminate how the code in asdf interacts with them.

There are at least two ways of outlining the design. One is to give high level overviews of the various modules and how they interact with other modules. The other is to illustrate how code is actually invoked in common operations, this often being much more informative on a practical level (at least some find that to be the case). This document will attempt to do both.

We will start with a high-level review of concepts and terms and point to where these are handled in the asdf modules.

Because of the complexity, this initial design overview will focus on issues of validation and tree construction when reading.

17.1 Construction in progress

Before we get into further details, a word on the transition to new plugin APIs. Starting in asdf 2.8 we've introduced new interfaces for extending the asdf library to support additional tags and schemas. The interfaces were redesigned with the following goals in mind:

- Simplify the connection between tags and their schema content. The old "resolver" system involves sending the tag URI through a lengthy series of transformations to get the filesystem path to the schema document. This has been error-prone and difficult to troubleshoot, so the new "resource mapping" system explicitly maps schema URIs to their content, and tag URIs directly to schema URIs.
- Make it easier to separate schemas from extension code. Until now the schemas have always been provided by the same Python package that implements support for their tags, but we would like to move the schemas to language-agnostic repositories that non-Python implementations can use. To better support this, the new interface splits the old extension plugin into two new plugins, one of which is dedicated to schemas.
- Allow tag serialization support to handle arbitrary sets of URIs. Previously tag code was restricted to working with tag URIs that were identical except for version. This presented a problem for the transition of URIs from stsci.edu to asdf-format.org, so the new interface allows for supporting diverse URIs with the same code.
- Improve the terminology used in the tag serialization support classes. The old `ExtensionType` has been renamed `Converter` to indicate its purpose, and to eliminate the ambiguity betwenen YAML types and Python types. The `to_tree` and `from_tree` methods have been renamed `to_yaml_tree` and `from_yaml_tree` to better indicate which tree they're expected to convert.

- Simplify the code and behavior of tag classes. Converters are used as instances instead of classes with a custom metaclass, Python sub-types are no longer automatically handled, URIs are treated as single values instead of broken down into various components, etc.

You can witness the gory details of this effort by clicking through the PR links on the asdf 2.8.0 roadmap.

Support for ASDF core tags has not yet been moved to the new system. Doing so would be a breaking change for users who subclass that code, so we'll need to wait until asdf 3.0 to do that.

17.2 Some terminology and definitions

URI vs URL (Universal Resource Identifier). This is distinguished from URL (Universal Resource Locator) primarily in that URI is a mechanism for a unique name that follows a particular syntax, but itself may not indicate where the resource is. Generally URLs are expected to be used on the web for the HTTP protocol, though for asdf, this isn't necessarily the case as mentioned next. Recent changes to the library permit use of URIs with the asdf:// scheme, which is intended to reduce confusion over the distinction between identifiers and locations.

Resolver: Tools to map URIs and tags into actual locations of schema files, which may be local directories (the usual approach) or an actual URL for retrieval over the network. This is more complicated that it may seem for reasons explained later. The resolver system has been deprecated in favor of resource mappings; new code should use the latter instead.

Global config: A global library configuration feature that was added in asdf 2.8. Allows plugins to be added or removed at runtime and `AsdfFile` defaults to be modified by the user. Accessed via the `get_config` method on the top-level `asdf` module. For example, the default ASDF Standard version for new files can be set like this:

```
asdf.get_config().default_version = "1.3.0"
```

Or a resource mapping plugin added at runtime like this:

```
asdf.get_config().add_resource_mapping({"http://somewhere.org/resources/foo", b"foo resource␣
↪content"})
```

Entry point: A Python packaging feature that allows asdf to use plugins provided by other packages. Entry points are registered when a package is installed and become available to asdf without any additional effort on the part of the user. See Entry points specification for more information.

Resource mapping: An asdf plugin that provides access to "resources" which are binary blobs associated with a URI. These resources are mostly schemas, but any resource may provided by a mapping. Resource mappings are provided via entry points or added at runtime using a method on the global config object. This feature is intended to replace the deprecated "resolver" mechanism.

Extension: An extension to the ASDF Standard that defines additional YAML tags. In the future an extension may include other additional features such as binary block compressors or filters, but currently only tags are supported.

Extension implementation: An asdf plugin that implements an extension to the ASDF Standard. This is the asdf library's support for an extended set of YAML tags. The library currently provides two interfaces for implementing extensions: the `AsdfExtension` class and the new, still-experimental `Extension` class. Extension implementations are provided via entry points or added at runtime using a method on the global config object. The `AsdfFile` also permits adding additional extensions on a per-instance basis, but use of that feature is discouraged and may be removed in asdf 3.0.

Tag code/tag class: A class responsible for converting a family of tags into Python objects and vice versa. Each extension implementation includes a list of such classes. For the original `AsdfExtension` API, the tag classes all implement the `ExtensionType` interface. For the new API, tag classes implement `Converter`.

Validator: Tool to confirm that the YAML conforms to the schemas that apply. A lot goes on in this area and it is pretty complex in the implementation.

Tree building: The YAML content is built into a tree in two stages. The YAML parser converts the raw YAML into a custom Python structure. It is that structure that is validated. Then if no errors are found, the tree is converted into a tree where tagged nodes get converted into corresponding Python objects (usually, an option exists to prevent this from happening, which is useful for some applications), e.g., WCS object or numpy arrays (well, not quite that simply for numpy arrays).

The above is a simplified view of what happens when an ASDF file is read.

Most of resolver tools and code is in `resolver.py` (but not all).

Most of the validation code is in `schema.py`.

The code that builds the trees is spread in many places: `tagged.py`, `treeutil.py`, `types.py` as well as all the extension code that supplies code to handle the tags within (and often the the associated schemas).

A note on the location of schemas and tag code; there is a bit of schizophrenic aspect to this since schema should be language agnostic and in that view, not bundled with specific language library code. But currently nearly all of the implementation is in Python so while the long-term goal is to keep them separate, it is more convenient to keep them together for now. You will see cases where they are separate and some where they are bundled. The introduction of a separate plugin for providing access to schemas (the "resource mapping") is intended to allow extension authors to keep the schema documents in a separate language-neutral repository.

17.3 Actions that happen when an AsdfFile is instantiated

The asdf plugins (new and old-style extensions as well as resource mappings) registered as entry points can be obtained by calling methods in `entry_points.py`. These methods are invoked by `config.AsdfConfig` the first time library needs to use the plugins, and thereafter are cached within that config object. Both extensions and resource mappings are stored wrapped in proxy objects (`ExtensionProxy` and `ResourceMappingProxy`, respectively) that carry additional metadata like the package name and version of the entry point, and add some convenience methods on top of what the extension developer provides. Additionally, `ExtensionProxy` allows the library to treat both new-style `Extension` instances and old-style `AsdfExtension` instances similarly.

To see the list of extensions loaded by the library, call `asdf.get_config().extensions`. To see the list of resource mappings, call `asdf.get_config().resource_mappings`. Both of these properties are lazy-loaded and then cached, so the first call will take a moment to complete but subsequent calls will return immediately.

When an `AsdfFile` class is instantiated, one thing that happens on the `__init__` is that `self._process_plugin_extensions()` is called. This method retrieves the extensions from the global config and selects those that are compatible with the `AsdfFile`'s ASDF Standard version. It returns the resulting list, which is assigned to the `_plugin_extensions` variable. The term "plugin extensions" constrasts with "user extensions" which are additional extensions provided by the user as an argument to `AsdfFile.__init__`.

The extension lists are used by `AsdfFile` to create the file's `ExtensionList` and `ExtensionManager` instances, which manage extensions for the old and new extension APIs, respectively. These instances are created lazily when the `extension_list` and `extension_manager` properties are first accessed, to help speed up the initial construction of the `AsdfFile`.

The `extension_manager` is responsible for mapping tag URIs to schema URIs for validation and retrieving type converters (instances of the `Converter` interface) by Python type or by YAML tag URI. `extension_list` handles the same duties, but for old-style extensions. `extension_manager` takes precedence over `extension_list` throughout the asdf library, so `extension_list` will only be consulted if `extension_manager` can't handle a particular tag or Python type.

17.4 On the subject of resolvers and tag/url mapping

The `AsdfFile` class has `tag_mapping` and `url_mapping` properties that each return the `extension_list` properties of the same name. These objects implement the original support for mapping tag URIs to schema content that in the new API is provided by resource mappings.

`tag_mapping` and `url_mapping` are each `resolver.Resolver` instances that are generated from the mapping lists in the old-style extensions. These lists consist of 2-tuples. In the first case it is a mechanism to map the tag string to a url string, typically with an expected prefix or suffix to the tag (suffix is typical) so that given a full tag, it generates a url that includes the suffix. This permits one mapping to cover many tag variants (The details of mapping machinery with examples are given in a later section since understanding this is essential to defining new tags and corresponding schemas).

The URL mapping works in a similar way, except that it consists of 2-tuples where the first element is the common elements of the url, and the second part maps it to an actual location (url or file path). Again the second part may include a place holder for the suffix or prefix, and code to generate the path to the schema file.

The use of the resolver object turns these lists into functions so that supplied the appropriate input that matches something in the list, it gives the corresponding output.

17.5 Outline of how an ASDF file is opened and read into the corresponding Python object.

The starting point can be found in `asdf.py` essentially through the following chain (many calls and steps left out to keep it simpler to follow)

When `asdf.open("myasdffile.asdf")` is called, it is aliased to `asdf.open_asdf` which first creates an instance of `asdf.AsdfFile` (let's call the instance `af`), then calls `af._open_impl()` and then `af._open_asdf`. That invokes a call to `generic_io.get_file()`.

`generic.py` basically contains code to handle all the variants of I/O possible (files, streaming, http access, etc). In this case it returns a `RealFile` instance that wraps a local file system file.

Next the file is examined to see if it is an ASDF file (first by examining the first few lines in the header). If it passes those checks, the header (yaml) section of the file is extracted through a proxy mechanism that signals an end of file when the end of the yaml is reached, but otherwise looks like a file object.

The yaml parsing phase described below normally returns a "tagged_tree". That is (somewhat simplified), it returns the data structure that yaml would normally return without any object conversion (i.e., all nodes are either dicts, lists, or scalar values), except that they are objects that now support a tag attribute that indicates if a tag was associated with that node and what the tag was.

This reader object is passed to the yaml parser by calling `yamlutil.load_tree`. A simple explanation for what goes on here is necessary to understand how this all works. Yaml supports various kinds of loaders. For security reasons, the "safe" loader is used (note that both C and python versions are supported through an indirection of the `_yaml_base_loader` defined at the beginning of that module that determines whether the C version is available). The loaders are recursive mechanisms that build the tree structure. Note that `yamlutil.load_tree` creates a temporary subclass of `AsdfLoader` and attaches a reference to the AsdfFile instance as the `.ctx` attribute of that temporary subclass.

One of the hooks that pyyaml supplies is the ability to overload the method `construct_object`. That's what the class `yamlutil.AsdfLoader` does. pyyaml calls this method at each node in the tree to see if anything special should be done. One could perform conversion to predefined objects here, but instead it does the following: it sees if the node.tag attribute is handled by yaml itself (examples?) it calls that constructor which returns the type yaml converts it to. Otherwise:

- it converts the node to the type indicated (dict, list, or scalar type) by yaml for that node.
- it obtains the appropriate tag class (an AsdfType subclass) from the AsdfFile instance (using `ctx.type_index.fix_yaml_tag` to deal with version issues to match the most appropriate tag class). The new extension API does not support this "fix YAML tag" feature so file's ExtensionManager is not used here.
- it wraps all the node alternatives in a special asdf `Tagged` class instance variant where that object contains a ._tag attribute that is a reference to the corresponding Tag class.

The loading process returns a tree of these Tagged object instances. This tagged_tree is then returned to the `af` instance (still running the `_open_asdf()` method) this tree is passed to to the `_validate()` method (This is the major reason that the tree isn't directly converted to an object tree since jsonschema would not be able to use the final object tree for validation, besides issues relate to the fact that things that don't validate may not be convertable to the designated object.)

The validate machinery is a bit confusing since there are essentially two basic approaches to how validation is done. One type of validation is for validation of schema files themselves, and the other for schemas for tags.

The schema.py file is fairly involved and the details are covered elsewhere. When the validator machinery is constructed, it uses the fundamental validation files (schemas). But this doesn't handle the fact that the file being validated is yaml, not json and that there are items in yaml not part of json so special handling is needed. And the way it is handled is through a internal mechanism of the jsonschema library. There is a method that jsonschema calls recursively for a validator and it is called iter_errors. The subclass of the jsonschema validator class is defined as schema.ASDFValidator and this method is overloaded in this class. Despite its name, it's primary purpose is to validate the special features that yaml has, namely applying schemas associated with tags (this is not part of the normal jsonschema scheme [ahem]). It is in this method that it looks for a tag for a node and if it exists and in the tag_index, loads the appropriate schema and applies it to the node. (jsonschemas are normally only associated with a whole json entity rather than specific nodes). While the purpose of this method is to iteratively handle errors that jsonschema detects, it has essentially been repurposed as the means of interjecting handling tag schemas.

In order to prevent repeated loading of the same schema, the lru caching scheme is used (from functools in the standard library) where the last n cached schemas are saved (details of how this works were recently changed to prevent a serious memory leak)

In any event, a lot is going on behind the scenes in validation and it deserves its own description elsewhere.

After validation, the tagged tree is then passed to yamlutil.tagged_tree_to_custom_tree() where the nodes in the tree that have special tag code convert the nodes into the appropriate Python objects that the base asdf and extensions are aware of. This is accomplished by that function defining a walker "callback" function (defined within that function as to pick up the af object intrinsically). The function then passes the callback walker to treeutil.walk_and_modify() where the tree will be traversed recursively applying the tag code associated with the tag to the more primitive tree representation replacing such nodes with Python objects. The tree traversal starts from the top, but the objects are created from the bottom up due to recursion (well, not quite that simple).

Understanding how this works is described more fully later on.

The result is what af.tree is set to, after doing another tree traversal looking for special type hooks for each node. It isn't clear if there is yet any use of that feature.

17.6 Not quite that simple

17.7 Outline of schema.py

This module is somewhat confusing due to the many functions and methods with some variant of validate in their name. This will try to make clear what they do (a renaming of these may be in order).

Here is a list of the functions/classes in `schema.py` and their purpose and where they sit in the order of things

default_ext_resolver

_type_to_tag: Handles mapping python types to yaml_tags, with the addition of support for OrderedDicts.

The next 5 functions are put in the `YAML_VALIDATORS` dictionary to ultimately be used by `_create_validator` to create the json validator object

---

validate_tag: Obtain the relevant tag for the supplied instance (either built ins or custom objects) and check that it matches the tag supplied to the function.

validate_propertyOrder: Not really a validator but rather as a trick to indicate that properties should retain their order.

validate_flowStyle: Not really a validator but rather as a trick to store what style to use to write the elements (for yaml objects and arrays)

validate_style: Not really a validator but rather as a trick to store info on what style to use to write the string.

validate_type: Used to deal with date strings

(It may make sense to rename the above to be more descriptive of the action than where they are stuck in the validation machinery; e.g., `set_propertyOrder`)

validate_fill_default: Set the default values for all properties that have a subschema that defines a default. Called indirectly in `fill_defaults`

validate_remove_default: does the opposite; remove all properties where value equals subschema default. Called indirectly in `remove_defaults` (For this and the above, validate in the name mostly confuses although it is used by the json validator.)

[these could be renamed as well since they do more than validate]

_create_validator: Creates an `ASDFValidator` class on the fly that uses the `jsonchema.validators` class created. This `ASDFValidator` class overrides the `iter_errors` method that is used to handle yaml tag cases (using the `._tag` attribute of the node to obtain the corresponding schema for that tag; e.g., it calls `load_schema` to obtain the right schema when called for each node in the jsonschema machinery). What isn't clear to me is why this is done on the fly and at least cached since it really only handles two variants of calls (basically which JSONSCHEMA version is to be used). Otherwise it doesn't appear to vary except for that. Admittedly, this is only created at the top level. This is called by `get_validator`.

class OrderedLoader: Inherits from the `_yaml_base_loader`, but otherwise does nothing new in the definition. But the following code defines `construct_mapping`, and then adds it as a method.

construct_mapping: Defined outside the `OrderedLoader` class but to be added to the `OrderedLoader` class by use of the base class add_constructor method. This function flattens the mapping and returns an `OrderedDict` of the property attributes (This needs some deep understanding of how the yaml parser actually works, which is not covered here. Apparently mappings can be represented as nested trees as the yaml is originally parsed. Or something like that.)

_load_schema: Loads json or yaml schemas (using the `OrderedLoader`).

_make_schema_loader: Defines the function load_schema using the provided resolver and _load_schema.

_make_resolver: Sets the schema loader for http, https, file, tag using a dictionary where these access methods are the keys and the schema loader returning only the schema (and not the uri). These all appear to use the same schema loader.

_load_draft4_metaschema:

load_custom_schema: Deals with custom schemas.

load_schema: Loads a schema from the specified location (this is cached). Called for every tag encountered (uses resolver machinery). Most of the complexity is in resolving json references. Calls `_make_schema_loader, resolver, reference.resolve_fragment, load_schema`

get_validator: Calls `_create_validator`. Is called by validate to return the created validator.

validate_large_literals: Ensures tree has no large literals (raises error if it does)

validate: Uses `get_validator` to get a validator object and then calls its validate method, and validates any large literals using `validate_large_literals`.

fill_defaults: Inserts attributes missing with the default value

remove_defaults: Where the tree has attributes with value equal to the default, strip the attribute.

check_schema: Checks schema against the metaschema.

Illustration of the where these are called:

`af._open_asdf` calls `af.validate` which calls `af._validate` which then calls `schema.validate` with the tagged tree as the first argument (it can be called again if there is a custom schema).

in schema.py

`validate -> get_validator -> _create_validator` (returns `ASDFValidator`). There are two levels of validation, those passed to the json_validation machinery for the schemas themselves, and those that the tag machinery triggers when the jsonschema validator calls through `iter_errors`. The first level handles all the tricks at the top. the `ASDFValidator` uses `load_schema` which in turn calls `_make_schema_loader`, then `_load_schema`. `_load_schema` uses the `OrderedLoader` to load the schemas.

Got that?

17.8 How the ASDF library works with pyyaml

17.8.1 A Tree Identifier

There are three flavors of trees in the process of reading ASDF files, one will see many references to each in the code and description below.

pyyaml native tree. This consists of standard Python containers like dict and list, and primitive values like string, integer, float, etc.

Tagged tree. These are similar to pyyaml native trees, but with the basic types wrapped in a class that has has an attribute that identifies the tag associated with that node so that later processing can apply the appropriate conversion code to convert to the final Python object.

Custom tree. This is a tree where all nodes are converted to the destination Python objects. For example, a numpy array or GWCS object.

17.8.2 Brief overview of how pyyaml constructs a Python tree

Understanding the process of creating Python objects from yaml requires some understanding of how pyyaml works. We will not go into all the details of pyyaml, but instead concentrate on one phase of its loading process. First an outline of the phases of processing that pyyaml goes through in loading a yaml file:

1. **scanning:** Converting the text into lexical tokens. Done in scanner.py
2. **parsing:** Converting the lexical tokens into parsing events. Done in parser.py.
3. **composing:** Converting the parsing events into a tree structure of pyyaml objects. Done in composer.py
4. **loading:** Converting the pyyaml tree into a Python object tree. Done in constructor.py

We will focus on the last step since that is where asdf integrates with how pyyaml works.

The key object in that module is `BaseConstructor` and its subclasses (asdf uses `SafeConstructor` for security purposes). Note that the pyyaml code is severely deficient in docstrings and comments. The key method that kicks off the conversion is `construct_document()`. Its responsibilities are to call the `construct_object()` method on the top node, "drain" any generators produced by construction (more on this later), and finally reset internal data structures once construction is complete.

The actual process seems somewhat mysterious because what is going on is that it is using generators in place of vanilla code to construct the children for mutable items. The general scheme is that each constructor for mutable elements (see as an example the `SafeConstructor.construct_yaml_seq()` method) is written as a generator that is expected to be asked a value twice. The first value returned is an empty object of the expected type (e.g., empty dict or list) and when asked a second time, it populates the previous object returned (and returns None, which is not used). (In rare exceptions, when called with `deep=True`, it does immediately populate the child nodes.)

Normally the generator is appended to the loader's state_generators attribute (a list) for later use. Any generators not handled in the recursive chain are handled when contruct_object returns to `construct_document`, where it iteratively asks each generator to complete populating its referenced object. Since that step of populating the object may in turn create new generators on the `state_generator` list, it only stops when no more generators appear on the list.

Why is this done? One reason is to handle references (anchors and aliases) that may be circular.

Suppose one had the following yaml source:

```
A: &a
    x: 1
    B:
        item1: 42
        item2: life, the universe, and everything
    circular: *a
```

Without generators, it would not be possible to handle this case since the node identified by anchor `a` has not been fully constructed when pyyaml encounters a reference to that anchor among the same node's descendants. The use of the generator allows creation of the container object to reference to before it is populated so that the above construction will work when constructing the tree. To follow the above example in more detail, the construction creates a dictionary for `a` and then returns to the `construct_document()` method, which then starts handling the generators put on the list (there is only one in this case). The generator then populates the contents of `a`. For the attribute `B` it encounters a new mutable container, and puts its generator on the list to handle, and then makes a reference to `a` which now is defined. One last time it handles the generator for `B` and since each item in that is not a container, the construction completes.

Pyyaml tracks pending objects in a recursive objects dict and throws an exception if generators fail to handle reference cycles. (The conversion of the tagged tree to the custom tree, performed later does not use the same technique; explained later)

17.8.3 How ASDF hooks into pyyaml construction

ASDF makes use of this by adding generators to this process by defining a new construct method `construct_undefined()` that handles all ASDF tag cases. This is added to the pyyaml dict of construct methods under the key of `None`. When pyyaml doesn't find a tag, that is what it uses as a key to handle unknown tags. Thus the construction is redirected to ASDF code. That code returns a generator in the case of mutable ASDF objects in line with how yaml works with mutable objects.

Historical note: Versions older than 2.6.0 did not work this way. Instead, those versions completely replaced the pyyaml method `construct_object()` with their own version that did not use generators as pyyaml did.

17.8.4 How conversion to ASDF objects is done

The current means of conversion is simpler to use by tag code, but also more subtle to understand how it actually works (for many, that means harder ;-)

The YAML loading process produces a tagged tree of basic Python types. The conversion of these into ASDF types is kicked off when the `AsdfFile` method `_open_asdf()` calls `yamlutil.tagged_tree_to_custom_tree()`. This function defines a walker function that is to be used with `treeutil.walk_and_modify()`. Most of what the walker function does is handle tag issues (e.g., can the tag be appropriately mapped to the tag creation code) and then returns the appropriate ASDF type by calling `tag_type.from_tree_tagged()`.

A note on tree traversal. One can traverse a tree in three ways: inorder, preorder, and postorder (`asdf.info()` uses a breadth-first traversal, yet another exciting option, which we won't describe here). These respectively mean whether nodes are visited in the horizontal ordering of the nodes displayed on a graphs (inorder), descending the tree from the root, doing the left node first, before the right node (preorder), or from the bottom up, doing both leaf nodes before the parent node (postorder). In generating the pyyaml tree, preorder works since it builds the tree from the root as one would expect in constructing the tree. But in converting the tagged tree into the custom tree, postorder is the natural course, where the children are generated first so that the parent node can refer to the final objects.

An important part of this conversion process is handled by an instance of the class `treeutil._TreeModificationContext`. This class does much the same trick that pyyaml does with generators. Although pyyaml creates references between basic python objects, these references must be converted to references between ASDF objects, and doing so requires a similar mechanism for building the ASDF objects. The `_TreeModificationContext` object (hereafter context object) holds the incomplete generators in a way similar to the pyyaml `construct_document` function.

There are differences though. The class `TreeModificationContext` provides methods to indicate if nodes are pending (i.e., incomplete), and there is a special value `PendingValue` that is a signal that the node hasn't been handled yet (e.g., it may be referencing something yet to be done). If `PendingValue` persists to the end, it indicates a failure to handle circular references in the tag code. This approach was taken because one of the earlier prototype implementations did something like this, passing dict and list subclasses that would throw an exception if a `PendingValue` element was accessed. That would have been more friendly to extension developers, but it was discarded because it wasn't thought it was worth turning all those high performance containers into slower asdf subclasses. We may want to revisit this if we decide to implement a tree that tracks "dirty" nodes and only writes to disk those that have changed, since in that case we'll need custom container subclasses anyway. We could also consider writing our own dict/list subclass in C so we could have our cake and eat it too.

The `walk_and_modify` code handles the case where the tag code returns a generator instead of a value. This generator is expected to be a similar kind of generator to what pyyaml uses, but differing in that instead of returning an empty container object it will populate whatever elements it can complete (e.g, all non-mutable ones), and complete the population of all the mutable members on the second iteration (which may, in turn, generate new generators for mutable elements contained within). When it detects a generator, the `walk_and_modify` code retrieves the first yielded value, then saves the generator in the context. When the top level of the context is reached (it handles nesting by indicating how many times it has been entered as a context), it starts "draining" the saved generators by doing the second iteration

on them. Like pyyaml, this second iteration may produce yet more generators that get saved, and thus keeps iterating on the saved generators until none are left.

It is not possible to construct reference cycles in immutable objects within pure Python code, and thus the generators are only needed for mutable constructs (e.g., dicts and lists).

Historical note: versions of the ASDF library prior to 2.6.0 required tag code when converting from a tagged object to a custom object to call `tagged_tree_to_custom_tree` on any values of attributes that may be arbitrarily nested objects. That no longer is needed with the latest code since any attribute that contains a mapping or sequence object automatically uses a generator, so population of that attribute is automatically deferred until the context is exited. Thus there is no need to explicitly call a function to populate it.

More explicitly, the `_recurse` function defined within `walk_and_modify` (in this postorder case) calls `_handle_children()` on the node in question first. If the node contains children, they are each fed back into `_recurse` and transformed into their final objects. A new node is populated with these transformed children, and that is the node that gets handed to `tag.from_tree_tagged()`. The effect is that the tag class receives a structure containing only transformed children, so it has no need to call `tagged_tree_to_custom_tree` on its own.

17.9 Future plans for SerializationContext

Currently, the `AsdfFile` itself is used as a container for serialization parameters and is passed to various methods in block.py, reference.py, schema.py, yamlutil.py, in `ExtensionType` subclasses, and others. This doesn't work very well for a couple of reasons. For one, the intention of `AsdfFile.write_to` is to "export" a copy of the file to disk without changing the in-memory `AsdfFile`, but since serialization parameters are read from the `AsdfFile`, the code currently modifies the open file as part of the write (and doesn't change it back). The second issue is that requiring an `AsdfFile` instance in so many method signatures forces the code (or users themselves) to create an empty dummy `AsdfFile` just to use the method.

The new `Converter` interface also accepts a `ctx` variable, but instead of an `AsdfFile` it's an instance of `SerializationContext`. This new object will serve the purpose of configuring serialization parameters and keeping necessary state, which means that the `AsdfFile` can go unmodified. The `SerializationContext` will be relatively lightweight and creating it will not incur as much of a performance penalty as creating an `AsdfFile`.

CHAPTER

EIGHTEEN

TAG VERSIONING AND YOU

Here we'll explore ASDF tag versioning, and walk through the process of supporting new and updated tags with AsdfType subclasses. AsdfType is the original API that is currently used to support the ASDF core tags. The new API, Converter, remains experimental and is currently (2020-09-24) being trialled in the asdf-astropy package.

18.1 ASDF versioning conventions

The ASDF Standard document provides a helpful overview of the various ASDF versioning conventions. We will be concerned with the *standard version* and individual *tag versions*.

18.2 Overview

The "standard version" or "ASDF Standard version" refers to the subset of individual tag versions that correspond to a specific release version of the ASDF Standard. The list of tags and versions is maintained in version_map files in the asdf-standard repository. For example, version_map-1.3.0.yaml contains a list of all tag versions that we must handle in order to fully support version 1.3.0 of the ASDF Standard. This list contains both "core" tags and non-core tags. The distinction there is that core tags are supported by this library, while the others are supported by some external Python library, such as astropy.

Our support for specific versions of the ASDF core tags is implemented with AsdfType subclasses. We'll discuss these more later, but for now the important thing to know is that each AsdfType class identifies the tag name and version(s) that it supports. Any core tag objects that lack this support will not serialize or deserialize properly.

When reading an ASDF file, the standard version doesn't play a significant role. Each core object is self-described by a YAML tag, which will be used to deserialize the object even if that tag conflicts with the overall standard version of the file. The library will use the tag to identify the most appropriate AsdfType to deserialize the object.

On write, the situation is different. The library may have a choice in which tag and/or AsdfType to use when serializing a given core object – if multiple versions of the same tag are present, which shall we choose? Here the standard version becomes important. The tag version selected is specified by the version map of the standard version that the file is being written under.

By default, the standard version used for writes is the latest offered, but users may override with another version.

18.3 Implementation details

18.3.1 Supported ASDF standard version list

The list of supported ASDF standard versions is maintained in `asdf.versioning.supported_versions`. The default version, `asdf.versioning.default_version`, is applied whenever a user declines to specify the standard version of a new file, and is set to the latest supported version.

18.3.2 AsdfType

In this library, each core tag is handled by a distinct `asdf.types.AsdfType` subclass. The AsdfType subclass is responsible for identifying the base name of its tag and the tag version(s) that it supports. It also provides any custom serialization/deserialization behavior that is required – AsdfType provides a default implementation that is only able to get and set attributes on dict-like objects.

In some cases, the AsdfType subclass also serves as the deserialized object type. For example, `asdf.types.core.Software` subclasses both AsdfType and dict. Its AsdfType-like behavior is to identify its tag and version, while its dict-like behavior is to act as a container for the attributes described by the tag. The class definition is mostly empty because as a dict it can rely on AsdfType's default implementation for (de)serialization.

Meanwhile, other AsdfType subclasses deserialize ASDF objects into instances of entirely separate classes. For example, `asdf.types.core.complex.ComplexType` handles complex number types, which aren't natively supported by YAML. ComplexType includes an additional class attribute, `types`, that lists the types that it is able to handle. It also provides custom implementations of the `to_tree` and `from_tree` class methods, which enable it to serialize a complex value into the appropriate string, and later rebuild the complex value from that string. This additional code is necessary because ComplexType does not (de)serialize itself.

We won't find an explicit list of AsdfType subclasses in the code; that list is assembled at runtime by AsdfType's metaclass, `asdf.types.AsdfTypeMeta`. The list can be inspected in the console like so:

```
>>> import asdf
>>> asdf.types._all_asdftypes
...
```

The AsdfType class attributes relevant to versioning are as follows:

- *name*: the base name of the tag, without its version string. For example, the tag URI `tag:stsci.edu:asdf/core/example-1.2.0` will have a name value of `"core/example"`.
- *version*: the primary tag version supported by the AsdfType. For the example above, version should be set to `"1.2.0"`. This should be the latest version that the tag supports.
- *supported_versions*: a set of tag versions that the AsdfType supports. In the above example, this might be `{"1.0.0", "1.1.0", "1.2.0"}`.

18.3.3 AsdfType selection rules

On read, the library will ideally be able to identify an AsdfType subclass that explicitly supports a given tag (either in the `version` class attribute or `supported_versions`. If that is not possible, it proceeds as follows:

- Use the AsdfType that supports the latest version that is less than the tag version. For example, if the tag is example-1.2.0, and AsdfType are available for 1.1.0 and 1.3.0, it will use the 1.1.0 subclass.
- If the above fails, use the earliest available AsdfType
- If no AsdfType exists that supports any version of that tag, then ASDF will deserialize the data into vanilla diff.

The library does not currently emit a warning in either of the first two cases, but in the third case, a warning is emitted.

The rules for selecting an AsdfType for a given tag are implemented by `asdf.type_index.AsdfTypeIndex.fix_yaml_tag`.

On write, the library will read the version map that corresponds to the ASDF Standard version in use, which dictates the subset of tag versions that are available. From the subset of AsdfType subclasses that handle those tag versions, it selects the subclass that is able to handle the type of the core object being serialized.

If an object is not supported by an AsdfType, its serialization will be handled by pyyaml. If pyyaml doesn't know how to serialize, it will raise `yaml.representer.RepresenterError`.

The rules for selecting an AsdfType for a given serializable object are implemented by `asdf.type_index.AsdfTypeIndex.from_custom_type`.

18.4 Implementing updates to the standard

Let's assume that there is a new standard version, 2.0.0, which includes one entirely new core tag, `core/new_object-1.0.0`, one backwards-compatible update to an existing tag, `core/updated_object-1.1.0`, and one breaking change to an existing tag, `core/breaking_object-2.0.0`. The following sections walk through the steps we'll need to take to support this new material.

18.4.1 Update the asdf-standard submodule commit pointer

The asdf-standard repository is integrated into the asdf repository as a submodule. To pull in new commits from the remote master (assumed to be named `origin`:

```
$ cd asdf-standard
$ git fetch origin
$ git checkout origin/master
```

18.4.2 Support the new standard version

The list can be found in `asdf.versioning.supported_versions`. Add `AsdfVersion("2.0.0")` to the end of the list (maintaining the sort order). This new version will become the default for new files, but we can update the definition of `asdf.versioning.default_version` if that is undesirable.

18.4.3 Support the new tag

Tags for previously unsupported objects are straightforward, since we don't need to worry about compatibility issues. Create a new AsdfType subclass with `name` and `version` set appropriately:

```
class NewObjectType(AsdfType):
    name = "core/new_object"
    version = "1.0.0"
```

In a real-life scenario, we'd need to actually support (de)serialization in some way, but those details are beyond the scope of this document.

18.4.4 Support the backwards-compatible tag

Since our updated_object-1.1.0 is backwards-compatible, we can share the same AsdfType subclass between it and the previous version. Presumably there exists an AsdfType that looks something like this:

```
class UpdatedObjectType(AsdfType):
    name = "core/updated_object"
    version = "1.0.0"
```

We'll need to update the version, and list 1.0.0 as a supported version, so that this class can continue to handle it:

```
class UpdatedObjectType(AsdfType):
    name = "core/updated_object"
    version = "1.1.0"
    supported_versions = {"1.0.0", "1.1.0"}
```

18.4.5 Support the breaking tag

The tag with breaking changes, core/breaking_object-2.0.0, may not be easily supported by the same AsdfType as the previous version. In that case, we can create a new AsdfType for 2.0.0, and as long as the two subclasses have distinct `version` values and non-overlapping `supported_versions` sets, they should coexist peaceably.

If this is the existing AsdfType:

```
class BreakingObjectType(AsdfType):
    name = "core/breaking_object"
    version = "1.0.0"
```

The new AsdfType might look something like this:

```
class BreakingObjectType2(AsdfType):
    name = "core/breaking_object"
    version = "2.0.0"
```

CAUTION: We might be tempted here to simply update the original BreakingObjectType, but failing to handle an older version of the tag constitutes dropping support for any ASDF Standard version that relies on that tag. This should only be done after a deprecation period and with a major version release of the library, since files written by an older release will not be readable by the new code.

Part V

Resources

CHAPTER

NINETEEN

CONTRIBUTING

We welcome feedback and contributions of all kinds. Contributions of code, documentation, or general feedback are all appreciated. This package follows the ASDF-format *Code Of Conduct* and strives to provide a welcoming community to all of our users and contributors.

New to GitHub or open source projects? If you are unsure about where to start or haven't used GitHub before, please feel free to contact the package maintainers.

Note: The ASDF Standard itself also has a repository on github. Suggestions for improvements to the ASDF Standard can be reported here.

19.1 Feedback, Feature Requests, and Bug Reports

Feedback, feature requests, and bug reports for the ASDF Python implementation can be posted via ASDF's github page. Please open a new issue any questions, bugs, feedback, or new features you would like to see. If there is an issue you would like to work on, please leave a comment and we will be happy to assist. New contributions and contributors are very welcome!

19.2 Contributing Code and Bug Fixes

To contribute code to ASDF please fork ASDF first and then open a pull request from your fork to ASDF. Typically, the main development work is done on the "master" branch. The rest of the branches are for release maintenance and should not be used normally. Unless otherwise told by a maintainer, pull request should be made and submitted to the "master" branch.

Note: The "stable" branch is protected and used for official releases.

We ask that all contributions include unit tests to verify that the code works as intended. These tests are run automatically by GitHub when pull requests are open. If you have difficulties with tests failing or writing new tests please reach out to the maintainers, who are glad to assist you.

Note: ASDF uses both `black` and `isort` to format your code, so we ask that you run these tools regularly on your code to ensure that it is formatted correctly.

To make this easier, we have included pre-commit support for ASDF. We suggest that you install `pre-commit`, so that your code is automatically formatted before you commit. For those who do not run these tools regularly, the `pre-commit-ci` bot will attempt to fix the issues with your pull request when you submit it.

CHAPTER

TWENTY

ASDF-FORMAT OPEN SOURCE CODE OF CONDUCT

We expect all "ASDF-format" organization projects to adopt a code of conduct that ensures a productive, respectful environment for all open source contributors and participants. We are committed to providing a strong and enforced code of conduct and expect everyone in our community to follow these guidelines when interacting with others in all forums. Our goal is to keep ours a positive, inclusive, successful, and growing community. The community of participants in open source Astronomy 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.

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 such as contacting conduct@stsci.edu (all emails sent to this address 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.

Parts of this code of conduct have been adapted from the Astropy and Numfocus codes of conduct.

CHAPTER

TWENTYONE

CHANGE LOG

21.1 2.12.1 (unreleased)

The ASDF Standard is at v1.6.0

- Overhaul of the ASDF documentation to make it more consistent and readable. [#1142, #1152]
- Update deprecated instances of `abstractproperty` to `abstractmethod` [#1148]
- Move build configuration into `pyproject.toml` [#1149, #1155]

21.2 2.12.0 (2022-06-06)

The ASDF Standard is at v1.6.0

- Added ability to display title as a comment in using the `info()` functionality. [#1138]
- Add ability to set asdf-standard version for schema example items. [#1143]

21.3 2.11.1 (2022-04-15)

The ASDF Standard is at v1.6.0

- Update minimum astropy version to 5.0.4. [#1133]

21.4 2.11.0 (2022-03-15)

The ASDF Standard is at v1.6.0

- Update minimum jsonschema version to 4.0.1. [#1105]

21.5 2.10.1 (2022-03-02)

The ASDF Standard is at v1.6.0

- Bugfix for circular build dependency for asdf. [#1094]
- Fix small bug with handling multiple schema uris per tag. [#1095]

21.6 2.10.0 (2022-02-17)

The ASDF Standard is at v1.6.0

- Replace asdf-standard submodule with pypi package. [#1079]

21.7 2.9.2 (2022-02-07)

The ASDF Standard is at v1.6.0

- Fix deprecation warnings stemming from the release of pytest 7.0.0. [#1075]
- Fix bug in pytest plugin when schemas are not in a directory named "schemas". [#1076]

21.8 2.9.1 (2022-02-03)

The ASDF Standard is at v1.6.0

- Fix typo in testing module `__init__.py` name. [#1071]

21.9 2.9.0 (2022-02-02)

The ASDF Standard is at v1.6.0

- Added the capability for tag classes to provide an interface to asdf info functionality to obtain information about the class attributes rather than appear as an opaque class object. [#1052 #1055]
- Fix tag listing when extension is not fully implemented. [#1034]
- Drop support for Python 3.6. [#1054]
- Adjustments to compression plugin tests and documentation. [#1053]
- Update setup.py to raise error if "git submodule update –init" has not been run. [#1057]
- Add ability for tags to correspond to multiple schema_uri, with an implied allOf among the schema_uris. [#1058, #1069]
- Add the URL of the file being parsed to `SerializationContext`. [#1065]
- Add `asdf.testing.helpers` module with simplified versions of test helpers previously available in `asdf.tests.helpers`. [#1067]

21.10 2.8.3 (2021-12-13)

The ASDF Standard is at v1.6.0

- Fix more use of 'python' where 'python3' is intended. [#1033]

21.11 2.8.2 (2021-12-06)

The ASDF Standard is at v1.6.0

- Update documentation to reflect new 2.8 features. [#998]
- Fix array compression for non-native byte order [#1010]
- Fix use of 'python' where 'python3' is intended. [#1026]
- Fix schema URI resolving when the URI prefix is also claimed by a legacy extension. [#1029]
- Remove 'name' and 'version' attributes from NDArrayType instances. [#1031]

21.12 2.8.1 (2021-06-09)

- Fix bug in block manager when a new block is added to an existing file without a block index. [#1000]

21.13 2.8.0 (2021-05-12)

The ASDF Standard is at v1.6.0

- Add `yaml_tag_handles` property to allow definition of custom yaml `%TAG` handles in the asdf file header. [#963]
- Add new resource mapping API for extending asdf with additional schemas. [#819, #828, #843, #846]
- Add global configuration mechanism. [#819, #839, #844, #847]
- Drop support for automatic serialization of subclass attributes. [#825]
- Support asdf:// as a URI scheme. [#854, #855]
- Include only extensions used during serialization in a file's metadata. [#848, #864]
- Drop support for Python 3.5. [#856]
- Add new extension API to support versioned extensions. [#850, #851, #853, #857, #874]
- Permit wildcard in tag validator URIs. [#858, #865]
- Implement support for ASDF Standard 1.6.0. This version of the standard limits mapping keys to string, integer, or boolean. [#866]
- Stop removing schema defaults for all ASDF Standard versions, and automatically fill defaults only for versions <= 1.5.0. [#860]
- Stop removing keys with `None` values from the tree on write. This fixes a long-standing issue where the tree structure is not preserved on write, but will break `ExtensionType` subclasses that depend on this behavior. Extension developers will need to modify their `to_tree` methods to check for `None` before adding a key to the tree (or modify the schema to permit nulls, if that is the intention). [#863]

- Deprecated the `auto_inline` argument to `AsdfFile.write_to` and `AsdfFile.update` and added `AsdfConfig.array_inline_threshold`. [#882, #991]
- Add `edit` subcommand to asdftool for efficient editing of the YAML portion of an ASDF file. [#873, #922]
- Increase limit on integer literals to signed 64-bit. [#894]
- Remove the `asdf.test` method and `asdf.__githash__` attribute. [#943]
- Add support for custom compression via extensions. [#931]
- Remove unnecessary `.tree` from search result paths. [#954]
- Drop support for bugs in older operating systems and Python versions. [#955]
- Add argument to `asdftool diff` that ignores tree nodes that match a JMESPath expression. [#956]
- Fix behavior of `exception` argument to `GenericFile.seek_until`. [#980]
- Fix issues in file type detection to allow non-seekable input and filenames without recognizable extensions. Remove the `asdf.asdf.is_asdf_file` function. [#978]
- Update `asdftool extensions` and `asdftool tags` to incorporate the new extension API. [#988]
- Add `AsdfSearchResult.replace` method for assigning new values to search results. [#981]
- Search for block index starting from end of file. Fixes rare bug when a data block contains a block index. [#990]
- Update asdf-standard to 1.6.0 tag. [#993]

21.14 2.7.5 (2021-06-09)

The ASDF Standard is at v1.5.0

- Fix bug in `asdf.schema.check_schema` causing relative references in metaschemas to be resolved incorrectly. [#987]
- Fix bug in block manager when a new block is added to an existing file without a block index. [#1000]

21.15 2.7.4 (2021-04-30)

The ASDF Standard is at v1.5.0

- Fix pytest plugin failure under older versions of pytest. [#934]
- Copy array views when the base array is non-contiguous. [#949]
- Prohibit views over FITS arrays that change dtype. [#952]
- Add support for HTTPS URLs and following redirects. [#971]
- Prevent astropy warnings in tests when opening known bad files. [#977]

21.16 2.7.3 (2021-02-25)

The ASDF Standard is at v1.5.0

- Add pytest plugin options to skip and xfail individual tests and xfail the unsupported ndarray-1.0.0 example. [#929]
- Fix bug resulting in invalid strides values for views over FITS arrays. [#930]

21.17 2.7.2 (2021-01-15)

The ASDF Standard is at v1.5.0

- Fix bug causing test collection failures in some environments. [#889]
- Fix bug when decompressing arrays with numpy 1.20. [#901, #909]

21.18 2.7.1 (2020-08-18)

The ASDF Standard is at v1.5.0

- Fix bug preventing access to copied array data after `AsdfFile` is closed. [#869]

21.19 2.7.0 (2020-07-23)

The ASDF Standard is at v1.5.0

- Fix bug preventing diff of files containing ndarray-1.0.0 objects in simplified form. [#786]
- Fix bug causing duplicate elements to appear when calling `copy.deepcopy` on a `TaggedList`. [#788]
- Improve validator performance by skipping unnecessary step of copying schema objects. [#784]
- Fix bug with `auto_inline` option where inline blocks are not converted to internal when they exceed the threshold. [#802]
- Fix misinterpretation of byte order of blocks stored in FITS files. [#810]
- Improve read performance by skipping unnecessary rebuild of tagged tree. [#787]
- Add option to `asdf.open` and `fits_embed.AsdfInFits.open` that disables validation on read. [#792]
- Fix bugs and code style found by adding F and W `flake8` checks. [#797]
- Eliminate warnings in pytest plugin by using `from_parent` when available. [#799]
- Prevent validation of empty tree when `AsdfFile` is initialized. [#794]
- All warnings now subclass `asdf.exceptions.AsdfWarning`. [#804]
- Improve warning message when falling back to an older schema, and note that fallback behavior will be removed in 3.0. [#806]
- Drop support for jsonschema 2.x. [#807]
- Stop traversing oneOf and anyOf combiners when filling or removing default values. [#811]
- Fix bug in version map caching that caused incompatible tags to be written under ASDF Standard 1.0.0. [#821]

- Fix bug that corrupted ndarrays when the underlying block array was converted to C order on write. [#827]
- Fix bug that produced unreadable ASDF files when an ndarray in the tree was both offset and broadcasted. [#827]
- Fix bug preventing validation of default values in `schema.check_schema`. [#785]
- Add option to disable validation of schema default values in the pytest plugin. [#831]
- Prevent errors when extension metadata contains additional properties. [#832]

21.20 2.6.0 (2020-04-22)

The ASDF Standard is at v1.5.0

- AsdfDeprecationWarning now subclasses DeprecationWarning. [#710]
- Resolve external references in custom schemas, and deprecate asdf.schema.load_custom_schema. [#738]
- Add `asdf.info` for displaying a summary of a tree, and `AsdfFile.search` for searching a tree. [#736]
- Add pytest plugin option to skip warning when a tag is unrecognized. [#771]
- Fix generic_io `read_blocks()` reading past the requested size [#773]
- Add support for ASDF Standard 1.5.0, which includes several new transform schemas. [#776]
- Enable validation and serialization of previously unhandled numpy scalar types. [#778]
- Fix handling of trees containing implicit internal references and reference cycles. Eliminate need to call `yamlutil.custom_tree_to_tagged_tree` and `yamlutil.tagged_tree_to_custom_tree` from extension code, and allow `ExtensionType` subclasses to return generators. [#777]
- Fix bug preventing history entries when a file was previously saved without them. [#779]
- Update developer overview documentation to describe design of changes to handle internal references and reference cycles. [#781]

21.21 2.5.2 (2020-02-28)

The ASDF Standard is at v1.4.0

- Add a developer overview document to help understand how ASDF works internally. Still a work in progress. [#730]
- Remove unnecessary dependency on six. [#739]
- Add developer documentation on schema versioning, additional schema and extension-related tests, and fix a variety of issues in `AsdfType` subclasses. [#750]
- Update asdf-standard to include schemas that were previously missing from 1.4.0 version maps. [#767]
- Simplify example in README.rst [#763]

21.22 2.5.1 (2020-01-07)

The ASDF Standard is at v1.4.0

- Fix bug in test causing failure when test suite is run against an installed asdf package. [#732]

21.23 2.5.0 (2019-12-23)

The ASDF Standard is at v1.4.0

- Added asdf-standard 1.4.0 to the list of supported versions. [#704]
- Fix load_schema LRU cache memory usage issue [#682]
- Add convenience method for fetching the default resolver [#682]
- `SpecItem` and `Spec` were deprecated in `semantic_version` and were replaced with `SimpleSpec`. [#715]
- Pinned the minimum required `semantic_version` to 2.8. [#715]
- Fix bug causing segfault after update of a memory-mapped file. [#716]

21.24 2.4.2 (2019-08-29)

The ASDF Standard is at v1.3.0

- Limit the version of `semantic_version` to <=2.6.0 to work around a Deprecation warning. [#700]

21.25 2.4.1 (2019-08-27)

The ASDF Standard is at v1.3.0

- Define the `in` operator for top-level `AsdfFile` objects. [#623]
- Overhaul packaging infrastructure. Remove use of `astropy_helpers`. [#670]
- Automatically register schema tester plugin. Do not enable schema tests by default. Add configuration setting and command line option to enable schema tests. [#676]
- Enable handling of subclasses of known custom types by using decorators for convenience. [#563]
- Add support for jsonschema 3.x. [#684]
- Fix bug in `NDArrayType.__len__`. It must be a method, not a property. [#673]

21.26 2.3.3 (2019-04-02)

The ASDF Standard is at v1.3.0

- Pass `ignore_unrecognized_tag` setting through to ASDF-in-FITS. [#650]
- Use `$schema` keyword if available to determine meta-schema to use when testing whether schemas themselves are valid. [#654]
- Take into account resolvers from installed extensions when loading schemas for validation. [#655]
- Fix compatibility issue with new release of `pyyaml` (version 5.1). [#662]
- Allow use of `pathlib.Path` objects for `custom_schema` option. [#663]

21.27 2.3.2 (2019-02-19)

The ASDF Standard is at v1.3.0

- Fix bug that occurs when comparing installed extension version with that found in file. [#641]

21.28 2.3.1 (2018-12-20)

The ASDF Standard is at v1.3.0

- Provide source information for `AsdfDeprecationWarning` that come from extensions from external packages. [#629]
- Ensure that top-level accesses to the tree outside a closed context handler result in an `OSError`. [#628]
- Fix the way `generic_io` handles URIs and paths on Windows. [#632]
- Fix bug in `asdftool` that prevented `extract` command from being visible. [#633]

21.29 2.3.0 (2018-11-28)

The ASDF Standard is at v1.3.0

- Storage of arbitrary precision integers is now provided by `asdf.IntegerType`. Reading a file with integer literals that are too large now causes only a warning instead of a validation error. This is to provide backwards compatibility for files that were created with a buggy version of ASDF (see #553 below). [#566]
- Remove WCS tags. These are now provided by the gwcs package. [#593]
- Deprecate the `asdf.asdftypes` module in favor of `asdf.types`. [#611]
- Support use of `pathlib.Path` with `asdf.open` and `AsdfFile.write_to`. [#617]
- Update ASDF Standard submodule to version 1.3.0.

21.30 2.2.1 (2018-11-15)

- Fix an issue with the README that caused sporadic installation failures and also prevented the long description from being rendered on pypi. [#607]

21.31 2.2.0 (2018-11-14)

- Add new parameter `lazy_load` to `AsdfFile.open`. It is `True` by default and preserves the default behavior. `False` detaches the loaded tree from the underlying file: all blocks are fully read and numpy arrays are materialized. Thus it becomes safe to close the file and continue using `AsdfFile.tree`. However, `copy_arrays` parameter is still effective and the active memory maps may still require the file to stay open in case `copy_arrays` is `False`. [#573]
- Add `AsdfConversionWarning` for failures to convert ASDF tree into custom types. This warning is converted to an error when using `assert_roundtrip_tree` for tests. [#583]
- Deprecate `asdf.AsdfFile.open` in favor of `asdf.open`. [#579]
- Add readonly protection to memory mapped arrays when the underlying file handle is readonly. [#579]

21.32 2.1.2 (2018-11-13)

- Make sure that all types corresponding to core tags are added to the type index before any others. This fixes a bug that was related to the way that subclass tags were overwritten by external extensions. [#598]

21.33 2.1.1 (2018-11-01)

- Make sure extension metadata is written even when constructing the ASDF tree on-the-fly. [#549]
- Fix large integer validation when storing `numpy` integer literals in the tree. [#553]
- Fix bug that caused subclass of external type to be serialized by the wrong tag. [#560]
- Fix bug that occurred when attempting to open invalid file but Astropy import fails while checking for ASDF-in-FITS. [#562]
- Fix bug that caused tree creation to fail when unable to locate a schema file for an unknown tag. This now simply causes a warning, and the offending node is converted to basic Python data structures. [#571]

21.34 2.1.0 (2018-09-25)

- Add API function for retrieving history entries. [#501]
- Store ASDF-in-FITS data inside a 1x1 BINTABLE HDU. [#519]
- Allow implicit conversion of `namedtuple` into serializable types. [#534]
- Fix bug that prevented use of ASDF-in-FITS with HDUs that have names with underscores. [#543]
- Add option to `generic_io.get_file` to close underlying file handle. [#544]
- Add top-level `keys` method to `AsdfFile` to access tree keys. [#545]

21.35 2.0.3 (2018-09-06)

- Update asdf-standard to reflect more stringent (and, consequently, more correct) requirements on the formatting of complex numbers. [#526]
- Fix bug with dangling file handle when using ASDF-in-FITS. [#533]
- Fix bug that prevented fortran-order arrays from being serialized properly. [#539]

21.36 2.0.2 (2018-07-27)

- Allow serialization of broadcasted `numpy` arrays. [#507]
- Fix bug that caused result of `set_array_compression` to be overwritten by `all_array_compression` argument to `write_to`. [#510]
- Add workaround for Python OSX write limit bug (see https://bugs.python.org/issue24658). [#521]
- Fix bug with custom schema validation when using out-of-line definitions in schema file. [#522]

21.37 2.0.1 (2018-05-08)

- Allow test suite to run even when package is not installed. [#502]

21.38 2.0.0 (2018-04-19)

- Astropy-specific tags have moved to Astropy core package. [#359]
- ICRSCoord tag has moved to Astropy core package. [#401]
- Remove support for Python 2. [#409]
- Create `pytest` plugin to be used for testing schema files. [#425]
- Add metadata about extensions used to create a file to the history section of the file itself. [#475]
- Remove hard dependency on Astropy. It is still required for testing, and for processing ASDF-in-FITS files. [#476]
- Add command for extracting ASDF extension from ASDF-in-FITS file and converting it to a pure ASDF file. [#477]
- Add command for removing ASDF extension from ASDF-in-FITS file. [#480]
- Add an `ExternalArrayReference` type for referencing arrays in external files. [#400]
- Improve the way URIs are detected for ASDF-in-FITS files in order to fix bug with reading gzipped ASDF-in-FITS files. [#416]
- Explicitly disallow access to entire tree for ASDF file objects that have been closed. [#407]
- Install and load extensions using `setuptools` entry points. [#384]
- Automatically initialize `asdf-standard` submodule in `setup.py`. [#398]
- Allow foreign tags to be resolved in schemas and files. Deprecate `tag_to_schema_resolver` property for `AsdfFile` and `AsdfExtensionList`. [#399]

- Fix bug that caused serialized FITS tables to be duplicated in embedded ASDF HDU. [#411]
- Create and use a new non-standard FITS extension instead of ImageHDU for storing ASDF files embedded in FITS. Explicitly remove support for the `.update` method of `AsdfInFits`, even though it didn't appear to be working previously. [#412]
- Allow package to be imported and used from source directory and builds in development mode. [#420]
- Add command to `asdftool` for querying installed extensions. [#418]
- Implement optional top-level validation pass using custom schema. This can be used to ensure that particular ASDF files follow custom conventions beyond those enforced by the standard. [#442]
- Remove restrictions affecting top-level attributes `data`, `wcs`, and `fits`. Bump top-level ASDF schema version to v1.1.0. [#444]

21.39 1.3.3 (2018-03-01)

- Update test infrastructure to rely on new Astropy v3.0 plugins. [#461]
- Disable use of 2to3. This was causing test failures on Debian builds. [#463]

21.40 1.3.2 (2018-02-22)

- Updates to allow this version of ASDF to be compatible with Astropy v3.0. [#450]
- Remove tests that are no longer relevant due to latest updates to Astropy's testing infrastructure. [#458]

21.41 1.3.1 (2017-11-02)

- Relax requirement on `semantic_version` version to 2.3.1. [#361]
- Fix bug when retrieving file format version from new ASDF file. [#365]
- Fix bug when duplicating inline arrays. [#370]
- Allow tag references using the tag URI scheme to be resolved in schema files. [#371]

21.42 1.3.0 (2017-10-24)

- Fixed a bug in reading data from an "http:" url. [#231]
- Implements v 1.1.0 of the asdf schemas. [#233]
- Added a function `is_asdf_file` which inspects the input and returns `True` or `False`. [#239]
- The `open` method of `AsdfInFits` now accepts URIs and open file handles in addition to HDULists. The `open` method of `AsdfFile` will now try to parse the given URI or file handle as `AsdfInFits` if it is not obviously a regular ASDF file. [#241]
- Updated WCS frame fields `obsgeoloc` and `obsgeovel` to reflect recent updates in `astropy` that changed representation from `Quantity` to `CartesianRepresentation`. Updated to reflect `astropy` change that combines `galcen_ra` and `galcen_dec` into `galcen_coord`. Added support for new field `galcen_v_sun`. Added support for required module versions for tag classes. [#244]

- Added support for `lz4` compression algorithm [#258]. Also added support for using a different compression algorithm for writing out a file than the one that was used for reading the file (e.g. to convert blocks to use a different compression algorithm) [#257]
- Tag classes may now use an optional `supported_versions` attribute to declare exclusive support for particular versions of the corresponding schema. If this attribute is omitted (as it is for most existing tag classes), the tag is assumed to be compatible with all versions of the corresponding schema. If `supported_versions` is provided, the tag class implementation can include code that is conditioned on the schema version. If an incompatible schema is encountered, or if deserialization of the tagged object fails with an exception, a raw Python data structure will be returned. [#272]
- Added option to `AsdfFile.open` to allow suppression of warning messages when mismatched schema versions are encountered. [#294]
- Added a diff tool to `asdftool` to allow for visual comparison of pairs of ASDF files. [#286]
- Added command to `asdftool` to display available tags. [#303]
- When possible, display name of ASDF file that caused version mismatch warning. [#306]
- Issue a warning when an unrecognized tag is encountered. [#295] This warning is silenced by default, but can be enabled with a parameter to the `AsdfFile` constructor, or to `AsdfFile.open`. Also added an option for ignoring warnings from unrecognized schema tags. [#319]
- Fix bug with loading JSON schemas in Python 3.5. [#317]
- Remove all remnants of support for Python 2.6. [#333]
- Fix issues with the type index used for writing out ASDF files. This ensures that items in the type index are not inadvertently overwritten by later versions of the same type. It also makes sure that schema example tests run against the correct version of the ASDF standard. [#350]
- Update time schema to reflect changes in astropy. This fixes an outstanding bug. [#343]
- Add `copy_arrays` option to `asdf.open` to control whether or not underlying array data should be memory mapped, if possible. [#355]
- Allow the tree to be accessed using top-level `__getitem__` and `__setitem__`. [#352]

21.43 1.2.1(2016-11-07)

- Make asdf conditionally dependent on the version of astropy to allow running it with older versions of astropy. [#228]

21.44 1.2.0(2016-10-04)

- Added Tabular model. [#214]
- Forced new blocks to be contiguous [#221]
- Rewrote code which tags complex objects [#223]
- Fixed version error message [#224]

21.45 1.0.5 (2016-06-28)

- Fixed a memory leak when reading wcs that grew memory to over 10 Gb. [#200]

21.46 1.0.4 (2016-05-25)

- Added wrapper class for astropy.core.Time, TaggedTime. [#198]

21.47 1.0.2 (2016-02-29)

- Renamed package to ASDF. [#190]
- Stopped support for Python 2.6 [#191]

21.48 1.0.1 (2016-01-08)

- Fixed installation from the source tarball on Python 3. [#187]
- Fixed error handling when opening ASDF files not supported by the current version of asdf. [#178]
- Fixed parse error that could occur sometimes when YAML data was read from a stream. [#183]

21.49 1.0.0 (2015-09-18)

- Initial release.

CHAPTER

TWENTYTWO

CITATION

If you use ASDF for work/research presented in a publication (whether directly, as a dependency to another package), please cite the Zenodo DOI for the appropriate version of ASDF. The versions (and their BibTeX entries) can be found at:

Zenodo DOI

https://zenodo.org/badge/latestdoi/18112754

We also recommend and encourage you to cite the general ASDF paper:

```
@article{GREENFIELD2015240,
title = {ASDF: A new data format for astronomy},
journal = {Astronomy and Computing},
volume = {12},
pages = {240-251},
year = {2015},
issn = {2213-1337},
doi = {https://doi.org/10.1016/j.ascom.2015.06.004},
url = {https://www.sciencedirect.com/science/article/pii/S2213133715000645},
author = {P. Greenfield and M. Droettboom and E. Bray},
keywords = {FITS, File formats, Standards, World coordinate system},
abstract = {We present the case for developing a successor format for the immensely␣
↪successful FITS format. We first review existing alternative formats and discuss why we do␣
↪not believe they provide an adequate solution. The proposed format is called the Advanced␣
↪Scientific Data Format (ASDF) and is based on an existing text format, YAML, that we␣
↪believe removes most of the current problems with the FITS format. An overview of the␣
↪capabilities of the new format is given along with specific examples. This format has the␣
↪advantage that it does not limit the size of attribute names (akin to FITS keyword names)␣
↪nor place restrictions on the size or type of values attributes have. Hierarchical␣
↪relationships are explicit in the syntax and require no special conventions. Finally, it␣
↪is capable of storing binary data within the file in its binary form. At its basic level,␣
↪the format proposed has much greater applicability than for just astronomical data.}
}
```

Part VI

See also

- The Advanced Scientific Data Format (ASDF) standard.
- `asdf` Python package distribution on pypi

Part VII

Index

- genindex
- modindex
- search

PYTHON MODULE INDEX

a

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

U

V

W

Y