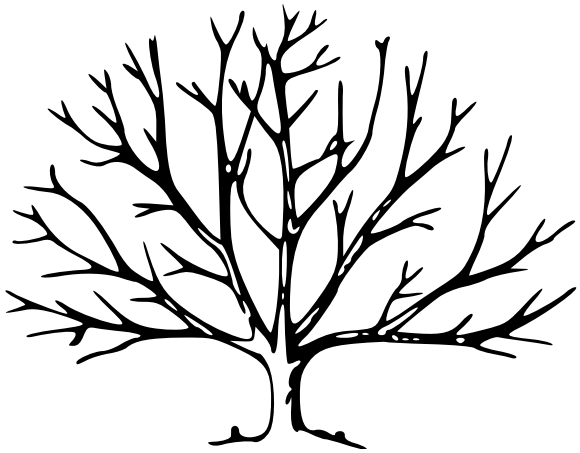


ASDF Standard

Release 1.6.0

The ASDF Developers

Apr 15, 2022

CONTENTS

This document describes the Advanced Scientific Data Format (ASDF), pronounced *AZ*-diff.

CHAPTER 1

INTRODUCTION

The Flexible Image Transport System (FITS) has been the de facto standard for storing and exchanging astronomical data for decades, but it is beginning to show its age. Developed in the late 1970s, the FITS authors made a number of implementation choices that, while common at the time, are now seen to limit its utility for the needs of modern science. As astronomy moves into a more varied set of data product types (data models) with richer and more complex metadata, FITS is being pushed to its breaking point. The issues with FITS are outlined in great detail in [Thomas2015].

Newer formats, such as VOTable (http://www.ivoa.net/documents/VOTable/) have partially addressed the problem of richer, more structured metadata, by using tree structures rather than flat key/value pairs. However, those text-based formats are unsuitable for storing large amounts of binary data. On the other end of the spectrum, formats such as HDF5 (http://www.hdfgroup.org/HDF5/) and BLZ (http://blaze.pydata.org/) address problems with large data sets and distributed computing, but don't really address the metadata needs of an interchange format. ASDF aims to exist in the same middle ground that made FITS so successful, by being a hybrid text and binary format: containing human editable metadata for interchange, and raw binary data that is fast to load and use. Unlike FITS, the metadata is highly structured and is designed up-front for extensibility.

ASDF has the following explicit goals:

- It has a hierarchical metadata structure, made up of basic dynamic data types such as strings, numbers, lists and mappings.
- It has human-readable metadata that can be edited directly in place in the file.
- The structure of the data can be automatically validated using schema.
- It's designed for extensibility: new conventions may be used without breaking backward compatibility with tools that do not understand those conventions. Versioning systems are used to prevent conflicting with alternative conventions.
- The binary array data (when compression is not used) is a raw memory dump, and techniques such as memory mapping can be used to efficiently access it.
- It is possible to read and write the file in as a stream, without requiring random access.
- It's built on top of industry standards, such as YAML (http://www.yaml.org) and JSON Schema (http://www.json-schema.org) to take advantage of a larger community working on the core problems

of data representation. This also makes it easier to support ASDF in new programming languages and environments by building on top of existing libraries.

- Since every ASDF file has the version of the specification to which it is written, it will be possible, through careful planning, to evolve the ASDF format over time, allowing for files that use new features while retaining backward compatibility with older tools.

ASDF is primarily intended as an interchange format for delivering products from instruments to scientists or between scientists. While it is reasonably efficient to work with and transfer, it may not be optimal for direct use on large data sets in distributed and high performance computing environments. That is explicitly not a goal of the ASDF standard, as those requirements can sometimes be at odds with the needs of an interchange format. ASDF still has a place in those environments as a delivery mechanism, even if it ultimately is not the actual format on which the computing is performed.

1.1 Implementations

The ASDF standard is being developed concurrently with a reference implementation written in Python (http://github.com/spacetelescope/asdf).

There are two prototype implementations for C++: asdf-cpp (https://github.com/spacetelescope/asdf-cpp) and asdf-cxx (https://github.com/eschnett/asdf-cxx). Neither is currently feature complete, but both provide enough functionality to read and write ASDF files.

There is also a work-in-progress wrapper (https://github.com/eschnett/asdf.jl) of the Python implementation for Julia (https://julialang.org).

1.2 Incorporated standards

The ASDF format is built on top of a number of existing standards:

- YAML 1.1 (http://yaml.org/spec/1.1/)
- JSON Schema Draft 4:
  - Core (http://tools.ietf.org/html/draft-zyp-json-schema-04)
  - Validation (http://tools.ietf.org/html/draft-fge-json-schema-validation-00)
  - Hyper-Schema (http://tools.ietf.org/html/draft-luff-json-hyper-schema-00)
- JSON Pointer (http://tools.ietf.org/html/rfc6901)
- Semantic Versioning 2.0.0 (http://semver.org/spec/v2.0.0.html)
- VOUnits (Units in the VO) (http://www.ivoa.net/documents/VOUnits/index.html)
- Zlib Deflate compression (http://www.zlib.net/feldspar.html)

CHAPTER 2

LOW-LEVEL FILE LAYOUT

The overall structure of a file is as follows (in order):

- *Header* (page 6)
- *Comments* (page 6), optional
- *Tree* (page 6), optional
- Zero or more *Blocks* (page 7)
- *Block index* (page 9), optional

ASDF is a hybrid text and binary format. The header, tree and block index are text, (specifically, in UTF-8 with DOS or UNIX-style newlines), while the blocks are raw binary.

The low-level file layout is designed in such a way that the tree section can be edited by hand, possibly changing its size, without requiring changes in other parts of the file. While such an operation may invalidate the *Block index* (page 9), the format is designed so that if the block index is removed or invalid, it may be regenerated by "skipping along" the blocks in the file.

The same is not true for resizing a block, which has an explicit size stored in the block header (except for, optionally, the last block).

Note also that, by design, an ASDF file containing no binary blocks is also a completely standard and valid YAML file.

Additionally, the spec allows for extra unallocated space after the tree and between blocks. This allows libraries to more easily update the files in place, since it allows expansion of certain areas without rewriting of the entire file.

2.1 Header

All ASDF files must start with a short one-line header. For example:

```
#ASDF 1.0.0
```

It is made up of two parts, separated by white space characters:

- **ASDF token**: The constant string `#ASDF`. This can be used to quickly identify the file as an ASDF file by reading the first 5 bytes. It begins with a # so it will be treated as a YAML comment such that the *Header* (page 6) and the *Tree* (page 6) together form a valid YAML file.
- **File format version**: The version of the low-level file format that this file was written with. This version may differ from the version of the ASDF specification, and is only updated when a change is made that affects the layout of file. It follows the Semantic Versioning 2.0.0 (http://semver.org/spec/v2.0.0.html) specification. See *Versioning Conventions* (page 17) for more information about these versions.

The header in EBNF form:

```
asdf_token = "#ASDF"
header     = asdf_token " " format_version ["\r"] "\n"
```

2.2 Comments

Additional comment lines may appear between the Header and the Tree.

The use of comments here is intended for information for the ASDF parser, and not information of general interest to the end user. All data of interest to the end user should be in the Tree.

Each line must begin with a # character.

2.3 Tree

The tree stores structured information using a subset of YAML Ain't Markup Language (YAML™) 1.1 (http://yaml.org/spec/1.1/) syntax (see *YAML subset* (page 11) for details on YAML features that are excluded from ASDF). While it is the main part of most ASDF files, it is entirely optional, and a ASDF file may skip it completely. This is useful for creating files in *Exploded form* (page 10). Interpreting the contents of this section is described in greater detail in *The tree in-depth* (page 11). This section only deals with the serialized representation of the tree, not its logical contents.

The tree is always encoded in UTF-8, without an explicit byteorder marker (BOM). Newlines in the tree may be either DOS (`"\r\n"`) or UNIX (`"\n"`) format.

In ASDF 1.6.0, the tree must be encoded in YAML version 1.1 (http://yaml.org/spec/1.1/). At the time of this writing, the latest version of the YAML specification is 1.2, however most YAML parsers only support YAML 1.1, and the benefits of YAML 1.2 are minor. Therefore, for maximum portability, ASDF requires that the YAML is encoded in YAML 1.1. To declare that YAML 1.1 is being used, the tree must begin with the following line:

```
%YAML 1.1
```

The tree must contain exactly one YAML document, starting with `---` (YAML document start marker) and ending with `...` (YAML document end marker), each on their own line. Between these two markers is the YAML content. For example:

```
%YAML 1.1
%TAG ! tag:stsci.edu:asdf/
--- !core/asdf-1.0.0
data: !core/ndarray-1.0.0
  source: 0
  datatype: float64
  shape: [1024, 1024]
...
```

The size of the tree is not explicitly specified in the file, so that it can easily be edited by hand. Therefore, ASDF parsers must search for the end of the tree by looking for the end-of-document marker (`...`) on its own line. For example, the following regular expression may be used to find the end of the tree:

```
\r?\n...\r?\n
```

Though not required, the tree should be followed by some unused space to allow for the tree to be updated and increased in size without performing an insertion operation in the file. It also may be desirable to align the start of the first block to a filesystem block boundary. This empty space may be filled with any content (as long as it doesn't contain the `block_magic_token` described in *Blocks* (page 7)). It is recommended that the content is made up of space characters (`0x20`) so it appears as empty space when viewing the file.

2.4 Blocks

Following the tree and some empty space, or immediately following the header, there are zero or more binary blocks.

Blocks represent a contiguous chunk of binary data and nothing more. Information about how to interpret the block, such as the data type or array shape, is stored entirely in `ndarray` structures in the tree, as described in *ndarray* (page 34). This allows for a very flexible type system on top of a very simple approach to memory management within the file. It also allows for new extensions to ASDF that might interpret the raw binary data in ways that are yet to be defined.

There may be an arbitrary amount of unused space between the end of the tree and the first block. To find the beginning of the first block, ASDF parsers should search from the end of the tree for the first occurrence of the `block_magic_token`. If the file contains no tree, the first block must begin immediately after the header with no padding.

2.4.1 Block header

Each block begins with the following header:

- `block_magic_token` (4 bytes): Indicates the start of the block. This allows the file to contain some unused space in which to grow the tree, and to perform consistency checks when jumping from one block to the next. It is made up of the following 4 8-bit characters:
  - in hexadecimal: d3, 42, 4c, 4b
  - in ascii: `"\323BLK"`
- `header_size` (16-bit unsigned integer, big-endian): Indicates the size of the remainder of the header (not including the length of the `header_size` entry itself or the `block_magic_token`), in bytes. It is stored explicitly in the header itself so that the header may be enlarged in a future version of the ASDF standard while retaining backward compatibility. Importantly, ASDF parsers should not assume a fixed size of the

header, but should obey the `header_size` defined in the file. In ASDF version 0.1, this should be at least 48, but may be larger, for example to align the beginning of the block content with a file system block boundary.

- `flags` (32-bit unsigned integer, big-endian): A bit field containing flags (described below).
- `compression` (4-byte byte string): The name of the compression algorithm, if any. Should be `\0\0\0\0` to indicate no compression. See *Compression* (page 8) for valid values.
- `allocated_size` (64-bit unsigned integer, big-endian): The amount of space allocated for the block (not including the header), in bytes.
- `used_size` (64-bit unsigned integer, big-endian): The amount of used space for the block on disk (not including the header), in bytes.
- `data_size` (64-bit unsigned integer, big-endian): The size of the block when decoded, in bytes. If `compression` is all zeros (indicating no compression), it **must** be equal to `used_size`. If compression is being used, this is the size of the decoded block data.
- `checksum` (16-byte string): An optional MD5 checksum of the used data in the block. The special value of all zeros indicates that no checksum verification should be performed.

2.4.2 Flags

The following bit flags are understood in the `flags` field:

- `STREAMED` (`0x1`): When set, the block is in streaming mode, and it extends to the end of the file. When set, the `allocated_size`, `used_size` and `data_size` fields are ignored. By necessity, any block with the `STREAMED` bit set must be the last block in the file.

2.4.3 Compression

Currently, two block compression types are supported:

- `zlib`: The zlib lossless compression algorithm. It is widely used, patent-unencumbered, and has an implementation released under a permissive license in zlib (http://www.zlib.net/).
- `bzp2`: The bzip2 lossless compression algorithm. It is widely used, assumed to be patent-unencumbered, and has an implementation released under a permissive license in the bzip2 library (http://www.bzip.org/).

2.4.4 Block content

Immediately following the block header, there are exactly `used_space` bytes of meaningful data, followed by `allocated_space - used_space` bytes of unused data. The exact content of the unused data is not enforced. The ability to have gaps of unused space allows an ASDF writer to reduce the number of disk operations when updating the file.

2.5 Block index

The block index allows for fast random access to each of the blocks in the file. It is completely optional: if not present, libraries may "skip along" the block headers to find the location of each block in the file. Libraries should detect invalid or obsolete block indices and ignore them and regenerate the index by skipping along the block headers.

The block index appears at the end of the file to make streaming an ASDF file possible without needing to determine the size of all blocks up front, which is non-trivial in the case of compression. It also allows for updating the index without an expensive insertion operation earlier in the file.

The block index must appear immediately after the allocated space for the last block in the file. If the last block is a streaming block, no block index may be present – the streaming block feature and block index are incompatible.

If no blocks are present in the file, the block index must also be absent.

The block index consists of a header, followed by a YAML document containing the indices of each block in the file.

The header must be exactly:

```
#ASDF BLOCK INDEX
```

followed by a DOS or UNIX newline.

Following the header is a YAML document (in YAML version 1.1, like the *Tree* (page 6)), containing a list of integers indicating the byte offset of each block in the file.

The following is an example block index:

```
#ASDF BLOCK INDEX
%YAML 1.1
--- [2043, 16340]
...
```

The offsets in the block index must be monotonically increasing, and must, by definition, be at least "block header size" apart. If they were allowed to appear in any order, it would be impossible to rebuild the index by skipping blocks were the index to become damaged or out-of-sync.

Additional zero-valued bytes may appear after the block index. This is mainly to support operating systems, such as Microsoft Windows, where truncating the file may not be easily possible.

2.5.1 Implementation recommendations

Libraries should look for the block index by reading backward from the end of the file.

Libraries should be conservative about what is an acceptable index, since addressing incorrect parts of the file could result in undefined behavior.

The following checks are recommended:

- Always ensure that the first offset entry matches the location of the first block in the file. This will catch the common use case where the YAML tree was edited by hand without updating the index. If they do not match, do not use the entire block index.
- Ensure that the last entry in the index refers to a block magic token, and that the end of the allocated space in the last block is immediately followed by the block index. If they do not match, do not use the entire block index.

- When using an offset in the block index, always ensure that the block magic token exists at that offset before reading data.

2.6 Exploded form

Exploded form expands a self-contained ASDF file into multiple files:

- An ASDF file containing only the header and tree, which by design is also a valid YAML file.
- *n* ASDF files, each containing a single block.

Exploded form is useful in the following scenarios:

- Not all text editors may handle the hybrid text and binary nature of the ASDF file, and therefore either can't open an ASDF file or would break an ASDF file upon saving. In this scenario, a user may explode the ASDF file, edit the YAML portion as a pure YAML file, and implode the parts back together.
- 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, not all web servers support this HTTP feature. 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.

Exploded form describes a convention for storing ASDF file content in multiple files, but it does not require any additions to the file format itself. There is nothing indicating that an ASDF file is in exploded form, other than the fact that some or all of its blocks come from external files. The exact way in which a file is exploded is up to the library and tools implementing the standard. In the simplest scenario, to explode a file, each *ndarray source property* (page 34) in the tree is converted from a local block reference into a relative URI.

CHAPTER 3

THE TREE IN-DEPTH

The ASDF tree, being encoded in YAML, is built out of the basic structures common to most dynamic languages: mappings (dictionaries), sequences (lists), and scalars (strings, integers, floating-point numbers, booleans, etc.). All of this comes "for free" by using YAML (http://yaml.org/spec/1.1/).

Since these core data structures on their own are so flexible, the ASDF standard includes a number of schema that define the structure of higher-level content. For instance, there is a schema that defines how *n-dimensional array data* (page 34) should be described. These schema are written in a language called *YAML Schema* (page 81) which is just a thin extension of JSON Schema, Draft 4 (http://json-schema.org/latest/json-schema-validation.html). (Such extensions are allowed and even encouraged by the JSON Schema standard, which defines the `$schema` attribute as a place to specify which extension is being used.) *ASDF Schemas* (page 19) contains an overview of how schemas are defined and used by ASDF. *ASDF Schema Definitions* (page 29) describes in detail all of the schemas provided by the ASDF Standard. reference to all of schemas in detail.

3.1 YAML subset

For reasons of portability, some features of YAML 1.1 are not permitted in an ASDF tree.

3.1.1 Restricted mapping keys

YAML itself places no restrictions on the object type used as a mapping key; floats, sequences, even mappings themselves can serve as a key. For example, the following is a perfectly valid YAML document:

```
%YAML 1.1
---
{foo: bar}:
  3.14159: baz
  [1, 2, 3]: qux
...
```

However, such a file may not be easily parsed in all languages. Python, for example, does not include a hashable mapping type, so the two major Python YAML libraries both fail to construct the object described by this document.

Floating-point keys are described as "not recommended" in the YAML 1.1 spec because YAML does not specify an accuracy for floats.

For these reasons, mapping keys in ASDF trees are restricted to the following scalar types:

- bool
- int
- str

3.2 Tags

YAML includes the ability to assign *Tags* (page 12) (or types) to any object in the tree. This is an important feature that sets it apart from other data representation languages, such as JSON. ASDF defines a number of custom tags, each of which has a corresponding schema. For example the tag of the root element of the tree must always be `tag:stsci.edu:asdf/core/asdf-1.1.0`, which corresponds to the *asdf schema* (page 29) –in other words, the top level schema for ASDF trees. A validating ASDF reader would encounter the tag when reading in the file, load the corresponding schema, and validate the content against it. An ASDF library may also use this information to convert to a native data type that presents a more convenient interface to the user than the structure of basic types stored in the YAML content.

For example:

```
%YAML 1.1
--- !<tag:stsci.edu:asdf/core/asdf-1.1.0>
data: !<tag:stsci.edu:asdf/core/ndarray-1.0.0>
  source: 0
  datatype: float64
  shape: [1024, 1024]
  byteorder: little
...
```

All tags defined in the ASDF standard itself begin with the prefix `tag:stsci.edu:asdf/`. This can be broken down as:

- `tag:` The standard prefix used for all YAML tags.
- `stsci.edu` The owner of the tag.
- `asdf` The name of the standard.

Following that is the "module" containing the schema (see *ASDF Schema Definitions* (page 29) for a list of the available modules). Lastly is the tag name itself, for example, `asdf` or `ndarray`. Since it is cumbersome to type out these long prefixes for every tag, it is recommended that ASDF files declare a prefix at the top of the YAML file and use it throughout. (Most standard YAML writing libraries have facilities to do this automatically.) For example, the following example is equivalent to the above example, but is more user-friendly. The `%TAG` declaration declares that the exclamation point (`!`) will be replaced with the prefix `tag:stsci.edu:asdf/`:

```
%YAML 1.1
%TAG ! tag:stsci.edu:asdf/
--- !core/asdf-1.1.0
data: !core/ndarray-1.0.0
  source: 0
  datatype: float64
  shape: [1024, 1024]
  byteorder: little
```

An ASDF parser may use the tag to look up the corresponding schema in the ASDF standard and validate the element. The schema definitions ship as part of the ASDF standard.

An ASDF parser may also use the tag information to convert the element to a native data type. For example, in Python, an ASDF parser may convert a *ndarray* (page 34) tag to a Numpy (http://www.numpy.org) array instance, providing a convenient and familiar interface to the user to access *n*-dimensional data.

The ASDF standard does not require parser implementations to validate or perform native type conversion, however. A parser may simply leave the tree represented in the low-level basic data structures. When writing an ASDF file, however, the elements in the tree must be appropriately tagged for other tools to make use of them.

ASDF parsers must not fail when encountering an unknown tag, but must simply retain the low-level data structure and the presence of the tag. This is important, as end users will likely want to store their own custom tags in ASDF files alongside the tags defined in the ASDF standard itself, and the file must still be readable by ASDF parsers that do not understand those tags.

3.3 References

It is possible to directly reference other items within the same tree or within the tree of another ASDF file. This functionality is based on two IETF standards: JSON Pointer (IETF RFC 6901) (http://tools.ietf.org/html/rfc6901) and JSON Reference (Draft 3) (http://tools.ietf.org/html/draft-pbryan-zyp-json-ref-03).

A reference is represented as a mapping (dictionary) with a single key/value pair. The key is always the special keyword `$ref` and the value is a URI. The URI may contain a fragment (the part following the # character) in JSON Pointer syntax that references a specific element within the external file. This is a `/`-delimited path where each element is a mapping key or an array index. If no fragment is present, the reference refers to the top of the tree.

Note: JSON Pointer is a very simple convention. The only wrinkle is that because the characters `'~'` (0x7E) and `'/'` (0x2F) have special meanings, `'~'` needs to be encoded as `'~0'` and `'/'` needs to be encoded as `'~1'` when these characters appear in a reference token.

When these references are resolved, this mapping should be treated as having the same logical content as the target of the URI, though the exact details of how this is performed is dependent on the implementation, i.e., a library may copy the target data into the source tree, or it may insert a proxy object that is lazily loaded at a later time.

For example, suppose we had a given ASDF file containing some shared reference data, available on a public webserver at the URI `http://www.nowhere.com/reference.asdf`:

```
wavelengths:
  - !core/ndarray
    source: 0
    shape: [256, 256]
    datatype: float
    byteorder: little
```

Another file may reference this data directly:

```
reference_data:
  $ref: "http://www.nowhere.com/reference.asdf#/wavelengths/0"
```

It is also possible to use references within the same file:

```
data: !core/ndarray
  source: 0
  shape: [256, 256]
  datatype: float
  byteorder: little
  mask:
    $ref: "#/my_mask"

my_mask: !core/ndarray
  source: 0
  shape: [256, 256]
  datatype: uint8
  byteorder: little
```

Reference resolution should be performed *after* the entire tree is read, therefore forward references within the same file are explicitly allowed.

Note: The YAML 1.1 standard itself also provides a method for internal references called "anchors" and "aliases". It does not, however, support external references. While ASDF does not explicitly disallow YAML anchors and aliases, since it explicitly supports all of YAML 1.1, their use is discouraged in favor of the more flexible JSON Pointer/JSON Reference standard described above.

3.4 Numeric literals

Integers represented as string literals in the ASDF tree must be no more than 64-bits. Due to ndarray types in numpy, this is further restricted to ranges defined for signed 64-bit integers (int64), not unsigned 64-bit integers (uint64).

3.5 Comments

It is quite common in FITS files to see comments that describe the purpose of the key/value pair. For example:

```
DATE    = '2015-02-12T23:08:51.191614' / Date this file was created (UTC)
TACID   = 'NOAO    '           / Time granting institution
```

Bringing this convention over to ASDF, one could imagine:

```
# Date this file was created (UTC)
creation_date: !time/utc
  2015-02-12T23:08:51.191614
# Time granting institution
time_granting_institution: NOAO
```

It should be obvious from the examples that these kinds of comments, describing the global meaning of a key, are much less necessary in ASDF. Since ASDF is not limited to 8-character keywords, the keywords themselves can be much more descriptive. But more importantly, the schema for a given key/value pair describes its purpose in detail. (It would be quite straightforward to build a tool that, given an entry in a YAML tree, looks up the schema's description associated with that entry.) Therefore, the use of comments to describe the global meaning of a value are strongly discouraged.

However, there still may be cases where a comment may be desired in ASDF, such as when a particular value is unusual or unexpected. The YAML standard includes a convention for comments, providing a handy way to include annotations in the ASDF file:

```
# We set this to filter B here, even though C is the more obvious
# choice, because B is handled with more accuracy by our software.
filter:
  type: B
```

Unfortunately, most YAML parsers will simply throw these comments out and do not provide any mechanism to retain them, so reading in an ASDF file, making some changes, and writing it out will remove all comments. Even if the YAML parser could be improved or extended to retain comments, the YAML standard does not define which values the comments are associated with. In the above example, it is only by standard reading conventions that we assume the comment is associated with the content following it. If we were to move the content, where should the comment go?

To provide a mechanism to add user comments without swimming upstream against the YAML standard, we recommend a convention for associating comments with objects (mappings) by using the reserved key name `//`. In this case, the above example would be rewritten as:

```
filter:
  //: |
    We set this to filter B here, even though C was used, because B
    is handled with more accuracy by our software.
  type: B
```

ASDF parsers must not interpret or react programmatically to these comment values: they are for human reference only. No schema may use `//` as a meaningful key.

3.6 Null values

YAML permits serialization of null values using the `null` literal:

```
some_key: null
```

Previous versions of the ASDF Standard were vague as to how nulls should be handled, and the Python reference implementation did not distinguish between keys with null values and keys that were missing altogether (and in fact, removed any keys assigned `None` from the tree on read or write). Beginning with ASDF Standard 1.6.0, ASDF implementatations are required to preserve keys even if assigned null values. This requirement does not extend back into previous versions, and users of the Python implementation should be advised that the YAML portion of a $< 1.6.0$ ASDF file containing null values may be modified in unexpected ways when read or written.

CHAPTER 4

VERSIONING CONVENTIONS

One of the explicit goals of ASDF is to be as future proof as possible. This involves being able to add features as needed while still allowing older libraries that may not understand those new features to reasonably make sense of the rest of the file.

The ASDF standard includes three categories of versions, all of which may advance independently of one another.

- **Standard version**: The version of the standard as a whole. This version provides a convenient handle to refer to a particular snapshot of the ASDF standard at a given time. This allows libraries to advertise support for "ASDF standard version X.Y.Z".
- **File format version**: Refers to the version of the blocking scheme and other details of the low-level file layout. This is the number that appears on the `#ASDF` header line at the start of every ASDF file and is essential to correctly interpreting the various parts of an ASDF file.
- **Schema versions**: Each schema for a particular YAML tag is individually versioned. This allows schemas to evolve, while still allowing data written to an older version of the schema to be validated correctly.

  Schemas provided by third parties (i.e. not in the ASDF specification itself) are also strongly encouraged to be versioned as well.

Version numbers all follow the same convention according to the Semantic Versioning 2.0.0 (http://semver.org/spec/v2.0.0.html) specification.

- **major version**: The major version number advances when a backward incompatible change is made. For example, this would happen when an existing property in a schema changes meaning. (An exception to this is that when the major version is 0, there are no guarantees of backward compatibility.)
- **minor version**: The minor version number advances when a backward compatible change is made. For example, this would happen when new properties are added to a schema.
- **patch version**: The patch version number advances when a minor change is made that does not directly affect the file format itself. For example, this would happen when a misspelling or grammatical error in the specification text is made that does not affect the interpretation of an ASDF file.
- **pre-release version**: An optional fourth part may also be present following a hyphen to indicate a pre-release version in development. For example, the pre-release of version `1.2.3` would be `1.2.3-dev+a2c4`.

4.1 Relationship of version numbers

The major number in the **standard version** is incremented whenever the major number in the **file format version** is incremented.

Schema versions are created and adjusted independently of the **standad version** and the **file format version**. New schemas are created with version `1.0.0` and are updated according to the Semantic Versioning conventions discussed above.

An update to any of the **schema versions** will be reflected in a bump of the **standard version** as well, although the version numbers will not necessarily match. Bumping a particular **schema version** will also require new versions of any of the schemas that make reference to it.

For example, schema `Foo` has version `1.0.0` in version `1.2.0` of the Standard. We make a backwards compatible change to `Foo` and bump its version to `1.1.0`. Schema `Bar` contains a reference to `Foo`. The current version of `Bar` is `1.1.0`, and we must now bump it to `1.2.0` to reflect the new reference to `Foo-1.1.0`. We also bump the Standard version to `1.3.0` to reflect the changes to these schemas.

4.2 Handling version mismatches

Given these conventions, the ASDF standard recommends certain behavior of ASDF libraries. ASDF libraries should, but are not required, to support as many existing versions of the file format and schemas as possible, and use the version numbers in the file to act accordingly.

For future-proofing, the library should gracefully handle version numbers that are greater than those understood by the library. The following applies to both kinds of version numbers that appear in the file: the **file format version** and **schema versions**.

- When encountering a **major version** that is greater than the understood version, by default, an exception should be raised. This behavior may be overridden through explicit user interaction, in which case the library will attempt to handle the element using the conventions of the most recent understood version.
- When encountering a **minor version** that is greater than the understood version, a warning should be emitted, and the library should attempt to handle the element using the conventions of the most recent understood version.
- When encountering a **patch version** that is greater than the understood version, silently ignore the difference and handle the element using the conventions of the most recent understood version.

When writing ASDF files, it is recommended that libraries provide both of the following modes of operation:

- Upgrade the file to the latest versions of the file format and schemas understood by the library.
- Preserve the version of the ASDF standard used by the input file.

Writing out a file that mixes versions of schema from different versions of the ASDF standard is not recommended, though such a file should be accepted by readers given the rules above.

CHAPTER 5

ASDF SCHEMAS

ASDF uses JSON Schema (http://json-schema.org) to perform validation of ASDF files. Schema validation of ASDF files serves the following purposes:

- Ensures conformity with core data types defined by the ASDF Standard. ASDF readers can detect whether an ASDF file has been modified in a way that would render it unreadable or unrecognizable.
- Enables interoperability between ASDF implementations. Implementations that recognize the same schema definitions should be able to interpret files containing instances of data types that conform to those schemas.
- Allows for the definition of custom data types. External software packages can provide ASDF schemas that correspond to types provided by that package, and then serialize instances of those types in a way that is standardized and portable.

All ASDF implementations must implement the types defined by the *core schemas* (page 29) and validate against them when reading files.[1] The ASDF Standard also defines two other categories of schemas, which are optional for ASDF implementations:

- *unit* (page 67)
- *time* (page 72)

The ASDF Standard also defines two metaschemas which are used to validate the ASDF schemas themselves:

- *YAML Schema* (page 81)
- *ASDF Schema* (page 88)

More information on the schemas defined by ASDF can be found in *ASDF Schema Definitions* (page 29).

[1] Implementations may expose the control of validation on reading to the user (e.g. to disable it on demand). However, validation on reading should be the default behavior.

5.1 Schema Implementation

ASDF schemas are encoded in YAML and conform to a superset of JSON Schema (http://json-schema.org) called *YAML Schema* (page 81). The version of YAML supported by ASDF is 1.1. Accordingly, all schemas begin with the following YAML header:

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

The following top-level attributes are required for all ASDF schemas:[2]

- `$schema`: Indicates the metaschema definition used to validate this schema
- `id`: A name that uniquely identifies the schema
- `tag`: The YAML tag corresponding to the type described by this schema

Each of these attributes is described in more detail below.

5.1.1 $schema

ASDF schemas use the top-level `$schema` attribute to declare the metaschema that is used to validate the schema itself. Most custom ASDF schemas will conform to *YAML Schema* (page 81) defined by the ASDF Standard, and so will have the following top-level attribute:

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

Some ASDF schemas use the *ASDF metaschema* (page 88) instead (e.g. *ndarray* (page 34)). It is also possible to create custom metaschemas, although these should always inherit from either YAML Schema or the ASDF metaschema.[3]

Some ASDF implementations may choose to validate the schemas themselves (e.g. as part of a regression testing suite). The `$schema` keyword should be used to determine the metaschema to be used for validation. All schemas should also validate successfully against *YAML Schema* (page 81).

5.1.2 id

The `id` represents the globally unique name of the schema. It must be a valid URI (https://tools.ietf.org/html/rfc3986) and cannot be an empty string or an empty fragment (e.g. #). See *Naming Conventions* (page 22) for conventions to ensure global uniqueness.

While the `id` must be a valid URI, it does not have to describe a real location on disk or on a network. For example, the `id` values for all schemas in the ASDF Standard begin with the prefix `http://stsci.edu/schemas/asdf/`. However, as of this writing, none of the schemas are actually hosted at that location.

The `id` keyword is used for reference resolution both within a schema and between schemas. Relative references within a schema are resolved against the `id` of that schema. A reference to an external schema uses the `id` of that schema. See *References* (page 21) below for additional information.

Each ASDF implementation must define how to resolve a schema `id` to a real resource that contains the schema itself. This could be done in a variety of ways, but the following seem like the most likely possibilities:

[2] The presence of `id` and `tag` is not currently enforced by the YAML Schema but may be in a future version of the ASDF Standard. Authors of new schemas should assume that at the very least `id` will be required in a future version of the Standard.

[3] For an example of how to inherit from another metaschema, look at the contents of the ASDF metaschema and see how there is a reference to the YAML schema in the top-level `allOf`.

- Resolve the `id` to a real network location (assuming the schema is actually hosted at that location)
- Map the `id` to a file location on disk that contains the schema

Other mappings are possible in theory. For example, a schema could be stored in a string literal as part of a program.

5.1.3 tag

The `tag` attribute is used by ASDF to associate an instance of a data type in an ASDF file with the appropriate schema to be used for validation. It is a concept from YAML (see the documentation (https://yaml.org/spec/1.1/#tag/information%20model)).

Libraries that provide custom schemas must ensure that the YAML tag that is written for a particular data type must match the `tag` attribute in the schema that corresponds to the data type. Tags must conform to the tag URI scheme which is defined in RFC 4151 (https://tools.ietf.org/html/rfc4151), but are otherwise perfectly arbitrary. However, certain *Naming Conventions* (page 22) are recommended in order to facilitate a mapping between `tag` and `id` attributes.

ASDF implementations must be able to map `tag` attributes to the corresponding schema `id`. The way that this mapping is defined is up to individual implementations. However, if the *Naming Conventions* (page 22) are followed, most implementations will be able to perform prefix matching and replacement.

While the `id` attribute will almost certainly become required in a future version of the ASDF Standard, the `tag` attribute may remain optional. This is because schemas can be referenced by `id` without necessarily referring to a particular tagged type in the YAML representation.

5.1.4 Descriptive information

Each schema may optionally contain descriptive fields: `title`, `description` and `examples`. These fields may contain core markdown syntax (which will be used for the purposes of rendering schema documentation by, for example, sphinx-asdf (https://github.com/spacetelescope/sphinx-asdf)).

- `title`: A one-line summary of the data type described by the schema
- `description`: A lengthier prose description of the schema
- `examples`: A list of example content that conforms to the schema, illustrating how to use it.

5.1.5 References

A particular ASDF schemas can contain references to other ASDF schemas. References are encoded by using the `$ref` attribute anywhere in the tree. While JSON Schema (http://json-schema.org) references are purely based on `id`, ASDF implementations must be able to resolve references using both `id` and `tag` attributes.

The resolution of `id` or `tag` references to actual schema files is up to individual implementations. It is recommended for ASDF implementations to use a two-phase mapping: one from `tag` to `id`, and another from `id` to an actual schema resource. In most cases, the `id` will be resolved to a location on disk (e.g. to a schema file that is installed in a known location). However, other scenarios might involve schemas that are hosted on a network, or schemas that are embedded in source files as string literals.

5.1.6 Naming Conventions

Schema `id` attributes must be valid URIs. Schema `tag` attributes must be valid URIs that conform to the tag URI scheme defined in RFC 4151 (https://tools.ietf.org/html/rfc4151) Aside from these requirements, assignment of these attributes is perfectly arbitrary. However, certain conventions are **strongly** recommended in order to ensure uniqueness and to enable a simple correspondence between the `id` and `tag` attributes. These conventions are described below.

All schema ids should encode the following information:

- **organization**: Indicates the organization that created the schema
- **standard**: The "standard" this schema belongs to. This will usually correspond to the name of the software package that provides this schema.
- **name**: The name of the data type corresponding to this schema.
- **version**: The version of the schema. See *Versioning Conventions* (page 17) for more details.

Consider the schemas from the ASDF Standard as an example. In this case, the **organization** is `stsci.edu`, which is the web address of the organization that created the schemas. The **standard** is `asdf`. Each individual schema in the ASDF Standard has a different **name** field. In the case of the *ndarray* (page 34) data type, for example, the name is `core/ndarray`. The version of *ndarray* (page 34) is `1.0.0`. Some other types in the ASDF Standard have multiple versions, such as `quantity-1.0.0` and *quantity-1.1.0* (page 70).

While schema ids can be any valid URI, under this convention they always begin with `http://`. The general format of the id attribute becomes:

```
http://<organization>/schemas/<standard>/<name>-<version>
```

Continuing with the example of *ndarray* (page 34), we have:

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

The idea behind the convention for `id` is that it should be possible (in principle if not in practice) for schemas to be hosted at the corresponding URL. This motivates the choice of the organization's web address as the **organization** component. However, this is not a requirement. The primary objective is to create a globally unique id.

Given the components defined above, the `tag` definition follows in a straightforward manner. The generic tag URI template is:

```
tag:<organization>:<standard>/<name>-<version>
```

Considering *ndarray* (page 34) once again, we have:

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

Following the naming convention for both `id` and `tag` attributes enables a simple mapping from `tag` to `id`. In this case, simply take the prefix `tag:stsci.edu:` and replace it with `http://stsci.edu/schemas/`.

5.2 Designing a new tag and schema

This section will walk through the development of a new tag and schema. In the example, suppose we work at the Example Organization, which can be found on the world wide web at `example.org`. We're developing a new instrument, `foo`, and we need a way to define the specialized metadata to describe the exposures that it will be generating.

According to the *Naming Conventions* (page 22), our `tag` and `id` attributes will consist of the following components:

- **organization**: `example.org`
- **standard**: `foo`
- **name**: `metadata`
- **version**: `1.0.0` (by convention the starting version for all new schemas)

So, for our example instrument metadata, the tag is:

```
tag:example.org:foo/metadata-1.0.0
```

Each tag should be associated with a schema in order to validate it. Each schema must also have a universally unique `id`, which is in the form of unique URI.

Note that this URI doesn't actually have to resolve to anything. In fact, visiting that URL in your web browser is likely to bring up a `404` error. All that's necessary is that it is universally unique and that the tool reading the ASDF file is able to map from a tag name to a schema URI, and then load the associated schema.

Again following with our example, we will assign the following URI to refer to our schema:

```
http://example.org/schemas/foo/metadata-1.0.0
```

Therefore, in our schema file, we have the following keys, one declaring the name of the YAML `tag`, and one defining the `id` of the schema:

```
id: "http://example.org/schemas/foo/metadata-1.0.0"
tag: "tag:example.org:foo/metadata-1.0.0"
```

Since our schema is just a basic ASDF schema, we will declare that it conforms to *YAML Schema* (page 81) defined by the ASDF Standard:

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

5.2.1 Descriptive information

Continuing our example, we include some *descriptive metadata* (page 21) about the data type declared by the schema itself:

```
title: |
  Metadata for the foo instrument.
description: |
  This stores some information about an exposure from the foo instrument.
examples:
  -
    - A minimal description of an exposure.
    - |
```

(continues on next page)

(continued from previous page)

```
        tag:example.org:foo/metadata-1.0.0
          exposure_time: 0.001
```

5.2.2 The schema proper

The rest of the schema describes the acceptable data types and their structure. The format used for this description comes straight out of JSON Schema, and rather than documenting all of the things it can do here, please refer to Understanding JSON Schema (http://spacetelescope.github.io/understanding-json-schema/), and the further resources available at json-schema.org (http://json-schema.org).

In our example, we'll define two metadata elements: the name of the investigator, and the exposure time, each of which also have a description:

```
type: object
properties:
  investigator:
    type: string
    description: |
      The name of the principal investigator who requested the
      exposure.

  exposure_time:
    type: number
    description: |
      The time of the exposure, in nanoseconds.
```

We'll also define an optional element for the exposure time unit. This is a somewhat contrived example to demonstrate how to include elements in your schema that are based on the custom types defined in the ASDF standard:

```
exposure_time_units:
  $ref: "http://stsci.edu/schemas/asdf/unit/unit-1.0.0"
  description: |
    The unit of the exposure time.
  default:
    s
```

Lastly, we'll declare `exposure_time` as being required, and allow extra elements to be added:

```
required: [exposure_time]
additionalProperties: true
```

5.2.3 The complete example

Here is our complete schema example:

```
%YAML 1.1
---
$schema: "http://stsci.edu/schemas/yaml-schema/draft-01"
id: "http://example.org/schemas/foo/metadata-1.0.0"
tag: "tag:example.org:foo/metadata-1.0.0"

title: |
  Metadata for the foo instrument.
description: |
  This stores some information about an exposure from the foo instrument.
examples:
  -
    - A minimal description of an exposure.
    - |
        tag:example.org:foo/metadata-1.0.0
          exposure_time: 0.001

type: object
properties:
  investigator:
    type: string
    description: |
      The name of the principal investigator who requested the
      exposure.

  exposure_time:
    type: number
    description: |
      The time of the exposure, in nanoseconds.

  exposure_time_units:
    $ref: "http://stsci.edu/schemas/asdf/unit/unit-1.0.0"
    description: |
      The unit of the exposure time.
    default:
      s

required: [exposure_time]
additionalProperties: true
```

5.3 Extending an existing schema

JSON Schema (http://json-schema.org) does not support the concept of inheritance, which makes it somewhat awkward to express type hierarchies. However, it is possible to create a custom schema that adds attributes to an existing schema (e.g. one defined in the ASDF Standard). It is important to remember that it is not possible to override or remove any of the attributes from the existing schema.

The following important caveats apply when extending an existing schema:

- It is not possible to redefine, override, or delete any attributes in the original schema.
- It will not be possible to add attributes to any node where the original schema declares `additionalProperties: false`
- Instances of the custom type will not be recognized as an instance of the original type when resolving schema references or processing YAML tags (i.e. there is no concept of polymorphism).

Here's an example of extending a schema using the *software* (page 54) schema defined by the ASDF Standard. Here's the original schema, for reference:

```
%YAML 1.1
---
$schema: "http://stsci.edu/schemas/yaml-schema/draft-01"
id: "http://stsci.edu/schemas/asdf/core/software-1.0.0"
title: |
  Describes a software package.
description: |
  General-purpose description of a software package.

tag: "tag:stsci.edu:asdf/core/software-1.0.0"
type: object
properties:
  name:
    description: |
      The name of the application or library.
    type: string

  author:
    description: |
      The author (or institution) that produced the software package.
    type: string

  homepage:
    description: |
      A URI to the homepage of the software.
    type: string
    format: uri

  version:
    description: |
      The version of the software used.  It is recommended, but not
      required, that this follows the (Semantic Versioning
      Specification)[http://semver.org/spec/v2.0.0.html].
    type: string
```

(continues on next page)

(continued from previous page)

```
required: [name, version]
additionalProperties: true
...
```

Since the software schema permits additional properties, we are free to extend it to include an email address for contacting the author:

```
%YAML 1.1
---
$schema: "http://stsci.edu/schemas/yaml-schema/draft-01"
id: "http://somewhere.org/schemas/software_extended-1.0.0"
title: |
  Describes a software package.
description: |
  Extension of ASDF core software schema to include the
  software author's contact email.

allOf:
  - $ref: http://stsci.edu/schemas/asdf/core/software-1.0.0
  - properties:
      author_email:
        description: |
          The contact email of the software author.
        type: string
    required: [author_email]
...
```

The crucial portion of this schema definition is the way that the `allOf` operator is used to join a reference to the base software schema with the definition of a new property called `author_email`.

The `allOf` combiner means that any instance that is validated against `software_extended-1.0.0` will have to conform to both the base software schema and the properties specific to the extended schema.

5.4 Default annotation

The JSON Schema spec includes a schema annotation attribute called `default` that can be used to describe the default value of a data attribute when that attribute is missing. Recent versions of the spec point out (http://json-schema.org/draft/2019-09/json-schema-core.html#rfc.section.7.7.1.1) that there is no single correct way to choose an annotation value when multiple are available due to references and combiners. This presents a problem when trying to fill in missing data in a file based on the schema `default`: if multiple conflicting values are available, the software does not know how to choose.

Previous versions of the ASDF Standard did not offer guidance on how to use `default`. The Python reference implementation read the first default that it encountered as a literal value and inserted that value into the tree when the corresponding attribute was otherwise missing. Until version 2.8, it also removed attributes on write whose values matched their schema defaults. The resulting files would appear to the casual viewer to be missing data, and may in fact be invalid against their schemas if the any of the removed attributes were required.

Implementations **must not** remove attributes with default values from the tree. Beginning with ASDF Standard 1.6.0, implementations also must not fill default values directly from the schema. This will avoid ambiguity when multiple schema defaults are present, and also permit the `default` attribute to contain a description that is not appropriate to use as a literal default value. For example:

```
default: An array of zeros matching the dimensions of the data array.
```

For ASDF Standard < 1.6.0, filling default values from the schema is required. This is necessary to support files written by older versions of the Python implementation.

CHAPTER 6

ASDF SCHEMA DEFINITIONS

This reference section describes the schema files for the built-in tags in ASDF.

ASDF schemas are arranged into "modules". All ASDF implementations must support the "core" module, but the other modules are optional.

6.1 Core

The `core` module contains schema that must be implemented by every asdf library.

6.1.1 core/asdf-1.1.0

Top-level schema for every ASDF file.

Description

This schema contains the top-level attributes for every ASDF file.

Outline

Schema Definitions

Internal Definitions

Original Schema

Schema Definitions

This type is an object with the following properties:

asdf_library

Describes the ASDF library that produced the file.

history

A log of transformations that have happened to the file. May include such things as data collection, data calibration pipelines, data analysis etc.

This node must validate against **any** of the following:

- array *No length restriction*

  The first 1 item in the list must be the following types:

  history_entry-1.0.0

- #/definitions/history-1.1.0

Internal Definitions

history-1.1.0

This type is an object with the following properties:

extensions *No length restriction*

The first 1 item in the list must be the following types:

extension_metadata-1.0.0

entries *No length restriction*

The first 1 item in the list must be the following types:

history_entry-1.0.0

Original Schema

```
%YAML 1.1
---
$schema: "http://stsci.edu/schemas/yaml-schema/draft-01"
id: "http://stsci.edu/schemas/asdf/core/asdf-1.1.0"
title: |
  Top-level schema for every ASDF file.

description: |
  This schema contains the top-level attributes for every ASDF file.

tag: "tag:stsci.edu:asdf/core/asdf-1.1.0"
type: object
properties:
  asdf_library:
    description: |
      Describes the ASDF library that produced the file.
    $ref: "software-1.0.0"

  history:
    description: |
      A log of transformations that have happened to the file.  May
      include such things as data collection, data calibration
      pipelines, data analysis etc.
    anyOf:
      # This is to support backwards compatibility with older history formats
      - type: array
        items:
          - $ref: "history_entry-1.0.0"
      # This is the new, richer history implementation that includes
```

(continues on next page)

(continued from previous page)

```
      # extension metadata.
      - $ref: "#/definitions/history-1.1.0"

additionalProperties: true
# Make sure that these two metadata fields are always at the top of the file
propertyOrder: [asdf_library, history]


# This contains the definition of the new history format, which includes
# metadata about the extensions used to create the file.
definitions:
  history-1.1.0:
    type: object
    properties:
      extensions:
        type: array
        items:
          - $ref: "extension_metadata-1.0.0"
      entries:
        type: array
        items:
          - $ref: "history_entry-1.0.0"
...
```

6.1.2 core/complex-1.0.0

Complex number value.

Description

Represents a complex number matching the following EBNF grammar

```
dot           = "."
plus-or-minus = "+" | "-"
digit         = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
sign          = "" | plus-or-minus
suffix        = "J" | "j" | "I" | "i"
inf           = "inf" | "INF"
nan           = "nan" | "NAN"
number        = digits | dot digits | digits dot digits
sci-suffix    = "e" | "E"
scientific    = number sci-suffix sign digits
real          = sign number | sign scientific
imag          = number suffix | scientific suffix
complex       = real | sign imag | real plus-or-minus imag
```

Though `J`, `j`, `I` and `i` must be supported on reading, it is recommended to use `i` on writing.

For historical reasons, it is necessary to accept as valid complex numbers that are surrounded by parenthesis.

Outline

Schema Definitions

Examples

Original Schema

Schema Definitions

string *No length restriction*

Must match the following pattern:

```
^(((((([+-]?(([0-9]+)|(\.[0-9]+)|([0-9]+\.[0-9]+)|(((inf)|(INF)))|(((nan)|(NAN)))))|([+-]?
 ↪(([0-9]+)|(\.[0-9]+)|([0-9]+\.[0-9]+)|(((inf)|(INF)))|(((nan)|(NAN))))[eE][+-]?[0-
 ↪9]+)))|([+-]?((((([0-9]+)|(\.[0-9]+)|([0-9]+\.[0-
 ↪9]+)|(((inf)|(INF)))|(((nan)|(NAN))))[iIjJ])|((([0-9]+)|(\.[0-9]+)|([0-9]+\.[0-
 ↪9]+)|(((inf)|(INF)))|(((nan)|(NAN))))[eE][+-]?[0-9]+[iIjJ]))))|((([+-]?(([0-9]+)|(\.[0-
 ↪9]+)|([0-9]+\.[0-9]+)|(((inf)|(INF)))|(((nan)|(NAN)))))|([+-]?(([0-9]+)|(\.[0-9]+)|([0-9]+\.
 ↪[0-9]+)|(((inf)|(INF)))|(((nan)|(NAN))))[eE][+-]?[0-9]+))[+-](((([0-9]+)|(\.[0-9]+)|([0-9]+\
 ↪.[0-9]+)|(((inf)|(INF)))|(((nan)|(NAN))))[iIjJ])|((([0-9]+)|(\.[0-9]+)|([0-9]+\.[0-
 ↪9]+)|(((inf)|(INF)))|(((nan)|(NAN))))[eE][+-]?[0-9]+[iIjJ])))))|(\(((((([+-]?(([0-9]+)|(\.[0-
 ↪9]+)|([0-9]+\.[0-9]+)|(((inf)|(INF)))|(((nan)|(NAN)))))|([+-]?(([0-9]+)|(\.[0-9]+)|([0-9]+\.
 ↪[0-9]+)|(((inf)|(INF)))|(((nan)|(NAN))))[eE][+-]?[0-9]+)))|([+-]?((((([0-9]+)|(\.[0-9]+)|([0-
 ↪9]+\.[0-9]+)|(((inf)|(INF)))|(((nan)|(NAN))))[iIjJ])|((([0-9]+)|(\.[0-9]+)|([0-9]+\.[0-
 ↪9]+)|(((inf)|(INF)))|(((nan)|(NAN))))[eE][+-]?[0-9]+[iIjJ]))))|((([+-]?(([0-9]+)|(\.[0-
 ↪9]+)|([0-9]+\.[0-9]+)|(((inf)|(INF)))|(((nan)|(NAN)))))|([+-]?(([0-9]+)|(\.[0-9]+)|([0-9]+\.
 ↪[0-9]+)|(((inf)|(INF)))|(((nan)|(NAN))))[eE][+-]?[0-9]+))[+-](((([0-9]+)|(\.[0-9]+)|([0-9]+\
 ↪.[0-9]+)|(((inf)|(INF)))|(((nan)|(NAN))))[iIjJ])|((([0-9]+)|(\.[0-9]+)|([0-9]+\.[0-
 ↪9]+)|(((inf)|(INF)))|(((nan)|(NAN))))[eE][+-]?[0-9]+[iIjJ])))))\)))$
```

Examples

1 real, -1 imaginary:

```
!core/complex-1.0.0 1-1j
```

0 real, 1 imaginary:

```
!core/complex-1.0.0 1J
```

-1 real, 0 imaginary:

```
!core/complex-1.0.0 -1
```

Original Schema

```
%YAML 1.1
---
$schema: "http://stsci.edu/schemas/yaml-schema/draft-01"
id: "http://stsci.edu/schemas/asdf/core/complex-1.0.0"
title: Complex number value.
description: |
  Represents a complex number matching the following EBNF grammar

  ```
    dot           = "."
    plus-or-minus = "+" | "-"
    digit         = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
    sign          = "" | plus-or-minus
    suffix        = "J" | "j" | "I" | "i"
```

(continues on next page)
```

(continued from previous page)

```
    inf             = "inf" | "INF"
    nan             = "nan" | "NAN"
    number          = digits | dot digits | digits dot digits
    sci-suffix      = "e" | "E"
    scientific      = number sci-suffix sign digits
    real            = sign number | sign scientific
    imag            = number suffix | scientific suffix
    complex         = real | sign imag | real plus-or-minus imag
  ```

  Though `J`, `j`, `I` and `i` must be supported on reading, it is
  recommended to use `i` on writing.

  For historical reasons, it is necessary to accept as valid complex numbers
  that are surrounded by parenthesis.

examples:
  -
    - 1 real, -1 imaginary
    - "!core/complex-1.0.0 1-1j"
  -
    - 0 real, 1 imaginary
    - "!core/complex-1.0.0 1J"
  -
    - -1 real, 0 imaginary
    - "!core/complex-1.0.0 -1"

tag: "tag:stsci.edu:asdf/core/complex-1.0.0"
type: string
This regex was automatically generated from a description of a grammar
pattern: "^(((((([+-]?(([0-9]+)|(\\.[0-9]+)|([0-9]+\\.[0-
 ↪9]+)|(((inf)|(INF)))|(((nan)|(NAN)))))|([+-]?(([0-9]+)|(\\.[0-9]+)|([0-9]+\\.[0-
 ↪9]+)|(((inf)|(INF)))|(((nan)|(NAN))))[eE][+-]?[0-9]+))))|([+-]?((((([0-9]+)|(\\.[0-9]+)|([0-
 ↪9]+\\.[0-9]+)|(((inf)|(INF)))|(((nan)|(NAN))))[iIjJ])|((((([0-9]+)|(\\.[0-9]+)|([0-9]+\\.[0-
 ↪9]+)|(((inf)|(INF)))|(((nan)|(NAN))))[eE][+-]?[0-9]+[iIjJ]))))|((((([+-]?(([0-9]+)|(\\.[0-
 ↪9]+)|([0-9]+\\.[0-9]+)|(((inf)|(INF)))|(((nan)|(NAN)))))|([+-]?(([0-9]+)|(\\.[0-9]+)|([0-
 ↪9]+\\.[0-9]+)|(((inf)|(INF)))|(((nan)|(NAN))))[eE][+-]?[0-9]+))[+-](((([0-9]+)|(\\.[0-
 ↪9]+)|([0-9]+\\.[0-9]+)|(((inf)|(INF)))|(((nan)|(NAN))))[iIjJ])|((((([0-9]+)|(\\.[0-9]+)|([0-
 ↪9]+\\.[0-9]+)|(((inf)|(INF)))|(((nan)|(NAN))))[eE][+-]?[0-9]+[iIjJ])))))|(\\((((((([+-]?(([0-
 ↪9]+)|(\\.[0-9]+)|([0-9]+\\.[0-9]+)|(((inf)|(INF)))|(((nan)|(NAN)))))|([+-]?(([0-9]+)|(\\.[0-
 ↪9]+)|([0-9]+\\.[0-9]+)|(((inf)|(INF)))|(((nan)|(NAN))))[eE][+-]?[0-9]+)))|([+-]?((((([0-
 ↪9]+)|(\\.[0-9]+)|([0-9]+\\.[0-9]+)|(((inf)|(INF)))|(((nan)|(NAN))))[iIjJ])|((((([0-9]+)|(\\.
 ↪[0-9]+)|([0-9]+\\.[0-9]+)|(((inf)|(INF)))|(((nan)|(NAN))))[eE][+-]?[0-9]+[iIjJ])))|(((([+-]?
 ↪(([0-9]+)|(\\.[0-9]+)|([0-9]+\\.[0-9]+)|(((inf)|(INF)))|(((nan)|(NAN)))))|([+-]?(([0-9]+)|(\
 ↪\.[0-9]+)|([0-9]+\\.[0-9]+)|(((inf)|(INF)))|(((nan)|(NAN))))[eE][+-]?[0-9]+))[+-](((([0-
 ↪9]+)|(\\.[0-9]+)|([0-9]+\\.[0-9]+)|(((inf)|(INF)))|(((nan)|(NAN))))[iIjJ])|((((([0-9]+)|(\\.
 ↪[0-9]+)|([0-9]+\\.[0-9]+)|(((inf)|(INF)))|(((nan)|(NAN))))[eE][+-]?[0-9]+[iIjJ]))))\\)))$"
...
```
```

6.1.3 core/ndarray-1.0.0

An *n*-dimensional array.

Description

There are two ways to store the data in an ndarray.

- Inline in the tree: This is recommended only for small arrays. In this case, the entire `ndarray` tag may be a nested list, in which case the type of the array is inferred from the content. (See the rules for type inference in the `inline-data` definition below.) The inline data may also be given in the `data` property, in which case it is possible to explicitly specify the `datatype` and other properties.
- External to the tree: The data comes from a *block* (page 7) within the same ASDF file or an external ASDF file referenced by a URI.

Outline

Schema Definitions

Examples

Internal Definitions

Original Schema

Schema Definitions

This node must validate against **any** of the following:

- #/definitions/inline-data
-

  This type is an object with the following properties:

  source

  The source of the data.

  If an integer: If positive, the zero-based index of the block within the same file. If negative, the index from the last block within the same file. For example, a source of `-1` corresponds to the last block in the same file.

  If a string, a URI to an external ASDF file containing the block data. Relative URIs and `file:` and `http:` protocols must be supported. Other protocols may be supported by specific library implementations.

  The ability to reference block data in an external ASDF file is intentionally limited to the first block in the external ASDF file, and is intended only to support the needs of *exploded* (page 10). For the more general case of referencing data in an external ASDF file, use tree *references* (page 13).

  This node must validate against **any** of the following:

  integer

  string *No length restriction*

  data

  The data for the array inline.

If `datatype` and/or `shape` are also provided, they must match the data here and can be used as a consistency check. `strides`, `offset` and `byteorder` are meaningless when `data` is provided.

shape

The shape of the array.

The first entry may be the string `*`, indicating that the length of the first index of the array will be automatically determined from the size of the block. This is used for streaming support. *No length restriction* Items in the array must be **any** of the following types:

integer

Minimum value: 0

This node has no type definition (unrestricted)

datatype

The data format of the array elements.

byteorder

The byte order (big- or little-endian) of the array data. *No length restriction*

Only the following values are valid for this node:

big

little

offset

The offset, in bytes, within the data for this start of this view.

Minimum value: 0

Default value: 0

strides

The number of bytes to skip in each dimension. If not provided, the array is assumed by be contiguous and in C order. If provided, must be the same length as the shape property. *No length restriction* Items in the array must be **any** of the following types:

integer

Minimum value: 1

integer

Maximum value: -1

mask

Describes how missing values in the array are stored. If a scalar number, that number is used to represent missing values. If an ndarray, the given array provides a mask, where non-zero values represent missing values in this array. The mask array must be broadcastable to the dimensions of this array.

This node must validate against **any** of the following:

number

complex-1.0.0

This node must validate against **all** of the following:

.

ndarray-1.0.0

· *This node has no type definition (unrestricted)*

Examples

An inline array, with implicit data type:

```
!core/ndarray-1.0.0
  [[1, 0, 0],
   [0, 1, 0],
   [0, 0, 1]]
```

An inline array, with an explicit data type:

```
!core/ndarray-1.0.0
  datatype: float64
  data:
    [[1, 0, 0],
     [0, 1, 0],
     [0, 0, 1]]
```

An inline structured array, where the types of each column are automatically detected:

```
!core/ndarray-1.0.0
  [[M110, 110, 205, And],
   [ M31,  31, 224, And],
   [ M32,  32, 221, And],
   [M103, 103, 581, Cas]]
```

An inline structured array, where the types of each column are explicitly specified:

```
!core/ndarray-1.0.0
  datatype: [['ascii', 4], uint16, uint16, ['ascii', 4]]
  data:
    [[M110, 110, 205, And],
     [ M31,  31, 224, And],
     [ M32,  32, 221, And],
     [M103, 103, 581, Cas]]
```

A double-precision array, in contiguous memory in a block within the same file:

```
!core/ndarray-1.0.0
  source: 0
  shape: [1024, 1024]
  datatype: float64
  byteorder: little
```

A view of a tile in that image:

```
!core/ndarray-1.0.0
  source: 0
  shape: [256, 256]
  datatype: float64
  byteorder: little
  strides: [8192, 8]
  offset: 2099200
```

A structured datatype, with nested columns for a coordinate in (*ra*, *dec*), and a 3x3 convolution kernel:

```
!core/ndarray-1.0.0
  source: 0
  shape: [64]
  datatype:
    - name: coordinate
      datatype:
        - name: ra
          datatype: float64
        - name: dec
          datatype: float64
    - name: kernel
      datatype: float32
      shape: [3, 3]
  byteorder: little
```

An array in Fortran order:

```
!core/ndarray-1.0.0
  source: 0
  shape: [1024, 1024]
  datatype: float64
  byteorder: little
  strides: [8192, 8]
```

An array where values of -999 are treated as missing:

```
!core/ndarray-1.0.0
  source: 0
  shape: [256, 256]
  datatype: float64
  byteorder: little
  mask: -999
```

An array where another array is used as a mask:

```
!core/ndarray-1.0.0
  source: 0
  shape: [256, 256]
  datatype: float64
  byteorder: little
  mask: !core/ndarray-1.0.0
    source: 1
    shape: [256, 256]
```

(continues on next page)

(continued from previous page)

```
    datatype: bool8
    byteorder: little
```

An array where the data is stored in the first block in another ASDF file.:

```
!core/ndarray-1.0.0
  source: external.asdf
  shape: [256, 256]
  datatype: float64
  byteorder: little
```

Internal Definitions

scalar-datatype

Describes the type of a single element.

There is a set of numeric types, each with a single identifier:

- `int8`, `int16`, `int32`, `int64`: Signed integer types, with the given bit size.
- `uint8`, `uint16`, `uint32`, `uint64`: Unsigned integer types, with the given bit size.
- `float32`: Single-precision floating-point type or “binary32”, as defined in IEEE 754.
- `float64`: Double-precision floating-point type or “binary64”, as defined in IEEE 754.
- `complex64`: Complex number where the real and imaginary parts are each single-precision floating-point (“binary32”) numbers, as defined in IEEE 754.
- `complex128`: Complex number where the real and imaginary parts are each double-precision floating-point (“binary64”) numbers, as defined in IEEE 754.

There are two distinct fixed-length string types, which must be indicated with a 2-element array where the first element is an identifier for the string type, and the second is a length:

- `ascii`: A string containing ASCII text (all codepoints < 128), where each character is 1 byte.
- `ucs4`: A string containing unicode text in the UCS-4 encoding, where each character is always 4 bytes long. Here the number of bytes used is 4 times the given length.

This node must validate against **any** of the following:

- string *No length restriction*

  Only the following values are valid for this node:

  - **int8**
  - **uint8**
  - **int16**
  - **uint16**
  - **int32**
  - **uint32**
  - **int64**
  - **uint64**

- **float32**
- **float64**
- **complex64**
- **complex128**
- **bool8**

- array *No length restriction*

  The first 2 items in the list must be the following types:

  string *No length restriction*

  Only the following values are valid for this node:

  ascii

  ucs4

  integer

  Minimum value: 0

datatype

The data format of the array elements. May be a single scalar datatype, or may be a nested list of datatypes. When a list, each field may have a name.

This node must validate against **any** of the following:

- #/definitions/scalar-datatype
- array *No length restriction* Items in the array must be **any** of the following types:
  - #/definitions/scalar-datatype
  - This type is an object with the following properties:

    name

    The name of the field *No length restriction*

    Must match the following pattern:

    ```
    [A-Za-z_][A-Za-z0-9_]*
    ```

    datatype

    byteorder

    The byteorder for the field. If not provided, the byteorder of the datatype as a whole will be used. *No length restriction*

    Only the following values are valid for this node:

    · **big**

    · **little**

    shape *No length restriction*

Items in the array are restricted to the following types:

integer

Minimum value: 0

inline-data

Inline data is stored in YAML format directly in the tree, rather than referencing a binary block. It is made out of nested lists.

If the datatype of the array is not specified, it is inferred from the array contents. Type inference is supported only for homogeneous arrays, not tables.

- If any of the elements in the array are YAML strings, the `datatype` of the entire array is `ucs4`, with the width of the largest string in the column, otherwise...
- If any of the elements in the array are complex numbers, the `datatype` of the entire column is `complex128`, otherwise...
- If any of the types in the column are numbers with a decimal point, the `datatype` of the entire column is `float64`, otherwise..
- If any of the types in the column are integers, the `datatype` of the entire column is `int64`, otherwise...
- The `datatype` of the entire column is `bool8`.

Masked values may be included in the array using `null`. If an explicit mask array is also provided, it takes precedence. *No length restriction* Items in the array must be **any** of the following types:

- number
- string *No length restriction*
- null
- complex-1.0.0
- #/definitions/inline-data
- boolean

Original Schema

```
%YAML 1.1
---
$schema: "http://stsci.edu/schemas/yaml-schema/draft-01"
id: "http://stsci.edu/schemas/asdf/core/ndarray-1.0.0"
tag: "tag:stsci.edu:asdf/core/ndarray-1.0.0"

title: >
  An *n*-dimensional array.

description: |
  There are two ways to store the data in an ndarray.
```

(continues on next page)

(continued from previous page)

```
  - Inline in the tree: This is recommended only for small arrays.  In
    this case, the entire ``ndarray`` tag may be a nested list, in
    which case the type of the array is inferred from the content.
    (See the rules for type inference in the ``inline-data``
    definition below.)  The inline data may also be given in the
    ``data`` property, in which case it is possible to explicitly
    specify the ``datatype`` and other properties.

  - External to the tree: The data comes from a [block](ref:block)
    within the same ASDF file or an external ASDF file referenced by a
    URI.

examples:
  -
    - An inline array, with implicit data type
    - |
        !core/ndarray-1.0.0
          [[1, 0, 0],
           [0, 1, 0],
           [0, 0, 1]]

  -
    - An inline array, with an explicit data type
    - |
        !core/ndarray-1.0.0
          datatype: float64
          data:
            [[1, 0, 0],
             [0, 1, 0],
             [0, 0, 1]]

  -
    - An inline structured array, where the types of each column are
      automatically detected
    - |
        !core/ndarray-1.0.0
          [[M110, 110, 205, And],
           [ M31,  31, 224, And],
           [ M32,  32, 221, And],
           [M103, 103, 581, Cas]]

  -
    - An inline structured array, where the types of each column are
      explicitly specified
    - |
        !core/ndarray-1.0.0
          datatype: [['ascii', 4], uint16, uint16, ['ascii', 4]]
          data:
            [[M110, 110, 205, And],
             [ M31,  31, 224, And],
             [ M32,  32, 221, And],
             [M103, 103, 581, Cas]]
```

(continues on next page)

(continued from previous page)

```
-
  - A double-precision array, in contiguous memory in a block within
    the same file
  - |
      !core/ndarray-1.0.0
        source: 0
        shape: [1024, 1024]
        datatype: float64
        byteorder: little

-
  - A view of a tile in that image
  - |
      !core/ndarray-1.0.0
        source: 0
        shape: [256, 256]
        datatype: float64
        byteorder: little
        strides: [8192, 8]
        offset: 2099200

-
  - A structured datatype, with nested columns for a coordinate in
    (*ra*, *dec*), and a 3x3 convolution kernel
  - |
      !core/ndarray-1.0.0
        source: 0
        shape: [64]
        datatype:
          - name: coordinate
            datatype:
              - name: ra
                datatype: float64
              - name: dec
                datatype: float64
          - name: kernel
            datatype: float32
            shape: [3, 3]
        byteorder: little

-
  - An array in Fortran order
  - |
      !core/ndarray-1.0.0
        source: 0
        shape: [1024, 1024]
        datatype: float64
        byteorder: little
        strides: [8192, 8]

-
```

(continues on next page)

(continued from previous page)

```
    - An array where values of -999 are treated as missing
    - |
        !core/ndarray-1.0.0
          source: 0
          shape: [256, 256]
          datatype: float64
          byteorder: little
          mask: -999

  -
    - An array where another array is used as a mask
    - |
        !core/ndarray-1.0.0
          source: 0
          shape: [256, 256]
          datatype: float64
          byteorder: little
          mask: !core/ndarray-1.0.0
            source: 1
            shape: [256, 256]
            datatype: bool8
            byteorder: little

  -
    - An array where the data is stored in the first block in
      another ASDF file.
    - |
        !core/ndarray-1.0.0
          source: external.asdf
          shape: [256, 256]
          datatype: float64
          byteorder: little

definitions:
  scalar-datatype:
    description: |
      Describes the type of a single element.

      There is a set of numeric types, each with a single identifier:

      - `int8`, `int16`, `int32`, `int64`: Signed integer types, with
        the given bit size.

      - `uint8`, `uint16`, `uint32`, `uint64`: Unsigned integer types,
        with the given bit size.

      - `float32`: Single-precision floating-point type or "binary32",
        as defined in IEEE 754.

      - `float64`: Double-precision floating-point type or "binary64",
        as defined in IEEE 754.
```

(continues on next page)

(continued from previous page)

```
    - `complex64`: Complex number where the real and imaginary parts
      are each single-precision floating-point ("binary32") numbers,
      as defined in IEEE 754.

    - `complex128`: Complex number where the real and imaginary
      parts are each double-precision floating-point ("binary64")
      numbers, as defined in IEEE 754.

    There are two distinct fixed-length string types, which must
    be indicated with a 2-element array where the first element is an
    identifier for the string type, and the second is a length:

    - `ascii`: A string containing ASCII text (all codepoints <
      128), where each character is 1 byte.

    - `ucs4`: A string containing unicode text in the UCS-4
      encoding, where each character is always 4 bytes long.  Here
      the number of bytes used is 4 times the given length.

  anyOf:
    - type: string
      enum: [int8, uint8, int16, uint16, int32, uint32, int64, uint64,
             float32, float64, complex64, complex128, bool8]
    - type: array
      items:
        - type: string
          enum: [ascii, ucs4]
        - type: integer
          minimum: 0
      minLength: 2
      maxLength: 2

datatype:
  description: |
    The data format of the array elements.  May be a single scalar
    datatype, or may be a nested list of datatypes.  When a list, each field
    may have a name.
  anyOf:
    - $ref: "#/definitions/scalar-datatype"
    - type: array
      items:
        anyOf:
          - $ref: "#/definitions/scalar-datatype"
          - type: object
            properties:
              name:
                type: string
                pattern: "[A-Za-z_][A-Za-z0-9_]*"
                description: The name of the field
              datatype:
                $ref: "#/definitions/datatype"
              byteorder:
```

(continues on next page)

(continued from previous page)

```
                type: string
                enum: [big, little]
                description: |
                  The byteorder for the field.  If not provided, the
                  byteorder of the datatype as a whole will be used.
              shape:
                type: array
                items:
                  type: integer
                  minimum: 0
            required: [datatype]

  inline-data:
    description: |
      Inline data is stored in YAML format directly in the tree, rather than
      referencing a binary block.  It is made out of nested lists.

      If the datatype of the array is not specified, it is inferred from
      the array contents.  Type inference is supported only for
      homogeneous arrays, not tables.

      - If any of the elements in the array are YAML strings, the
        `datatype` of the entire array is `ucs4`, with the width of
        the largest string in the column, otherwise...

      - If any of the elements in the array are complex numbers, the
        `datatype` of the entire column is `complex128`, otherwise...

      - If any of the types in the column are numbers with a decimal
        point, the `datatype` of the entire column is `float64`,
        otherwise..

      - If any of the types in the column are integers, the `datatype`
        of the entire column is `int64`, otherwise...

      - The `datatype` of the entire column is `bool8`.

      Masked values may be included in the array using `null`.  If an
      explicit mask array is also provided, it takes precedence.

    type: array
    items:
      anyOf:
        - type: number
        - type: string
        - type: "null"
        - $ref: "complex-1.0.0"
        - $ref: "#/definitions/inline-data"
        - type: boolean

anyOf:
  - $ref: "#/definitions/inline-data"
```

(continues on next page)

(continued from previous page)

```
- type: object
  properties:
    source:
      description: |
        The source of the data.

        - If an integer: If positive, the zero-based index of the
          block within the same file. If negative, the index from
          the last block within the same file.  For example, a
          source of `-1` corresponds to the last block in the same
          file.

        - If a string, a URI to an external ASDF file containing the
          block data.  Relative URIs and ``file:`` and ``http:``
          protocols must be supported.  Other protocols may be supported
          by specific library implementations.

        The ability to reference block data in an external ASDF file
        is intentionally limited to the first block in the external
        ASDF file, and is intended only to support the needs of
        [exploded](ref:exploded).  For the more general case of
        referencing data in an external ASDF file, use tree
        [references](ref:references).

      anyOf:
        - type: integer
        - type: string
          format: uri

    data:
      description: |
        The data for the array inline.

        If `datatype` and/or `shape` are also provided, they must
        match the data here and can be used as a consistency check.
        `strides`, `offset` and `byteorder` are meaningless when
        `data` is provided.
      $ref: "#/definitions/inline-data"

    shape:
      description: |
        The shape of the array.

        The first entry may be the string `*`, indicating that the
        length of the first index of the array will be automatically
        determined from the size of the block.  This is used for
        streaming support.
      type: array
      items:
        anyOf:
          - type: integer
            minimum: 0
```

(continues on next page)

(continued from previous page)

```
        - enum: ['*']

  datatype:
    description: |
      The data format of the array elements.
    $ref: "#/definitions/datatype"

  byteorder:
    description: >
      The byte order (big- or little-endian) of the array data.
    type: string
    enum: [big, little]

  offset:
    description: >
      The offset, in bytes, within the data for this start of this
      view.
    type: integer
    minimum: 0
    default: 0

  strides:
    description: >
      The number of bytes to skip in each dimension.  If not provided,
      the array is assumed by be contiguous and in C order.  If
      provided, must be the same length as the shape property.
    type: array
    items:
      anyOf:
        - type: integer
          minimum: 1
        - type: integer
          maximum: -1

  mask:
    description: >
      Describes how missing values in the array are stored.  If a
      scalar number, that number is used to represent missing values.
      If an ndarray, the given array provides a mask, where non-zero
      values represent missing values in this array.  The mask array
      must be broadcastable to the dimensions of this array.
    anyOf:
      - type: number
      - $ref: "complex-1.0.0"
      - allOf:
        - $ref: "ndarray-1.0.0"
        - datatype: bool8

dependencies:
  source: [shape, datatype, byteorder]

propertyOrder: [source, data, mask, datatype, byteorder, shape, offset, strides]
```

(continues on next page)

(continued from previous page)

```
...
```

6.1.4 core/table-1.0.0

A table.

Description

A table is represented as a list of columns, where each entry is a *column* (page 52) object, containing the data and some additional information.

The data itself may be stored inline as text, or in binary in either row- or column-major order by use of the `strides` property on the individual column arrays.

Each column in the table must have the same first (slowest moving) dimension.

Outline

Schema Definitions

Examples

Original Schema

Schema Definitions

This type is an object with the following properties:

columns

A list of columns in the table. *No length restriction*

Items in the array are restricted to the following types:

column-1.0.0

meta

Additional free-form metadata about the table.

object

Default value:

```
{}
```

Examples

A table stored in column-major order, with each column in a separate block:

```
!core/table-1.0.0
  columns:
  - !core/column-1.0.0
    data: !core/ndarray-1.0.0
      source: 0
      datatype: float64
      byteorder: little
      shape: [3]
    description: RA
    meta: {foo: bar}
```

(continues on next page)

(continued from previous page)

```
    name: a
    unit: !unit/unit-1.0.0 deg
  - !core/column-1.0.0
    data: !core/ndarray-1.0.0
      source: 1
      datatype: float64
      byteorder: little
      shape: [3]
    description: DEC
    name: b
  - !core/column-1.0.0
    data: !core/ndarray-1.0.0
      source: 2
      datatype: [ascii, 1]
      byteorder: big
      shape: [3]
    description: The target name
    name: c
```

A table stored in row-major order, all stored in the same block:

```
!core/table-1.0.0
  columns:
  - !core/column-1.0.0
    data: !core/ndarray-1.0.0
      source: 0
      datatype: float64
      byteorder: little
      shape: [3]
      strides: [13]
    description: RA
    meta: {foo: bar}
    name: a
    unit: !unit/unit-1.0.0 deg
  - !core/column-1.0.0
    data: !core/ndarray-1.0.0
      source: 0
      datatype: float64
      byteorder: little
      shape: [3]
      offset: 4
      strides: [13]
    description: DEC
    name: b
  - !core/column-1.0.0
    data: !core/ndarray-1.0.0
      source: 0
      datatype: [ascii, 1]
      byteorder: big
      shape: [3]
      offset: 12
      strides: [13]
```

(continues on next page)

(continued from previous page)

```
    description: The target name
    name: c
```

Original Schema

```
%YAML 1.1
---
$schema: "http://stsci.edu/schemas/yaml-schema/draft-01"
id: "http://stsci.edu/schemas/asdf/core/table-1.0.0"
tag: "tag:stsci.edu:asdf/core/table-1.0.0"

title: >
  A table.

description: |
  A table is represented as a list of columns, where each entry is a
  [column](ref:core/column-1.0.0)
  object, containing the data and some additional information.

  The data itself may be stored inline as text, or in binary in either
  row- or column-major order by use of the `strides` property on the
  individual column arrays.

  Each column in the table must have the same first (slowest moving)
  dimension.

examples:
  -
    - A table stored in column-major order, with each column in a separate block
    - |
        !core/table-1.0.0
          columns:
          - !core/column-1.0.0
            data: !core/ndarray-1.0.0
              source: 0
              datatype: float64
              byteorder: little
              shape: [3]
            description: RA
            meta: {foo: bar}
            name: a
            unit: !unit/unit-1.0.0 deg
          - !core/column-1.0.0
            data: !core/ndarray-1.0.0
              source: 1
              datatype: float64
              byteorder: little
              shape: [3]
            description: DEC
            name: b
          - !core/column-1.0.0
            data: !core/ndarray-1.0.0
```

(continues on next page)

(continued from previous page)

```
            source: 2
            datatype: [ascii, 1]
            byteorder: big
            shape: [3]
          description: The target name
          name: c

-
  - A table stored in row-major order, all stored in the same block
  - |
      !core/table-1.0.0
        columns:
        - !core/column-1.0.0
          data: !core/ndarray-1.0.0
            source: 0
            datatype: float64
            byteorder: little
            shape: [3]
            strides: [13]
          description: RA
          meta: {foo: bar}
          name: a
          unit: !unit/unit-1.0.0 deg
        - !core/column-1.0.0
          data: !core/ndarray-1.0.0
            source: 0
            datatype: float64
            byteorder: little
            shape: [3]
            offset: 4
            strides: [13]
          description: DEC
          name: b
        - !core/column-1.0.0
          data: !core/ndarray-1.0.0
            source: 0
            datatype: [ascii, 1]
            byteorder: big
            shape: [3]
            offset: 12
            strides: [13]
          description: The target name
          name: c

type: object
properties:
  columns:
    description: |
      A list of columns in the table.
    type: array
    items:
      $ref: column-1.0.0
```

(continues on next page)

(continued from previous page)

```
  meta:
    description: |
      Additional free-form metadata about the table.
    type: object
    default: {}

additionalProperties: false
required: [columns]
...
```

6.1.5 core/column-1.0.0

A column in a table.

Description

Each column contains a name and an array of data, and an optional description and unit.

Outline

Schema Definitions

Original Schema

Schema Definitions

This type is an object with the following properties:

name

The name of the column. Each name in a table (http://stsci.edu/schemas/asdf/core/table-1.0.0) must be unique. *No length restriction*

Must match the following pattern:

```
[A-Za-z_][A-Za-z0-9_]*
```

data

The array data for the column.

This node must validate against **all** of the following:

- ndarray-1.0.0

description

An optional description of the column. *No length restriction*

Default value: ‘’

unit

An optional unit for the column.

This node must validate against **all** of the following:

- ../unit/unit-1.0.0

meta

Additional free-form metadata about the column.

object

Default value:

```
{}
```

Original Schema

```yaml
%YAML 1.1
---
$schema: "http://stsci.edu/schemas/yaml-schema/draft-01"
id: "http://stsci.edu/schemas/asdf/core/column-1.0.0"
tag: "tag:stsci.edu:asdf/core/column-1.0.0"

title: >
  A column in a table.

description: |
  Each column contains a name and an array of data, and an optional description
  and unit.

type: object
properties:
  name:
    description: |
      The name of the column.  Each name in a
      [table](http://stsci.edu/schemas/asdf/core/table-1.0.0) must be
      unique.
    type: string
    pattern: "[A-Za-z_][A-Za-z0-9_]*"

  data:
    description: |
      The array data for the column.
    allOf:
      - $ref: ndarray-1.0.0

  description:
    description: |
      An optional description of the column.
    type: string
    default: ''

  unit:
    description:
      An optional unit for the column.
    allOf:
      - $ref: ../unit/unit-1.0.0

  meta:
    description:
      Additional free-form metadata about the column.
    type: object
    default: {}
```

(continues on next page)

(continued from previous page)

```
required: [name, data]
additionalProperties: false
...
```

6.1.6 core/constant-1.0.0

Specify that a value is a constant.

Description

Used as a utility to indicate that value is a literal constant.

Outline

Schema Definitions

Original Schema

Schema Definitions

tag:stsci.edu:asdf/core/constant-1.0.0

Original Schema

```
%YAML 1.1
---
$schema: "http://stsci.edu/schemas/yaml-schema/draft-01"
id: "http://stsci.edu/schemas/asdf/core/constant-1.0.0"
tag: "tag:stsci.edu:asdf/core/constant-1.0.0"
title: Specify that a value is a constant.
description: |
  Used as a utility to indicate that value is a literal constant.
...
```

6.1.7 core/software-1.0.0

Describes a software package.

Description

General-purpose description of a software package.

Outline

Schema Definitions

Original Schema

Schema Definitions

This type is an object with the following properties:

name

The name of the application or library. *No length restriction*

author

The author (or institution) that produced the software package. *No length restriction*

homepage

A URI to the homepage of the software. *No length restriction*

version

The version of the software used. It is recommended, but not required, that this follows the (Semantic Versioning Specification)[http://semver.org/spec/v2.0.0.html]. *No length restriction*

Original Schema

```
%YAML 1.1
---
$schema: "http://stsci.edu/schemas/yaml-schema/draft-01"
id: "http://stsci.edu/schemas/asdf/core/software-1.0.0"
title: |
  Describes a software package.
description: |
  General-purpose description of a software package.

tag: "tag:stsci.edu:asdf/core/software-1.0.0"
type: object
properties:
  name:
    description: |
      The name of the application or library.
    type: string

  author:
    description: |
      The author (or institution) that produced the software package.
    type: string

  homepage:
    description: |
      A URI to the homepage of the software.
    type: string
    format: uri

  version:
    description: |
      The version of the software used.  It is recommended, but not
      required, that this follows the (Semantic Versioning
      Specification)[http://semver.org/spec/v2.0.0.html].
    type: string

required: [name, version]
additionalProperties: true
...
```

6.1.8 core/history_entry-1.0.0

An entry in the file history.

Description

A record of an operation that has been performed upon a file.

Outline

Schema Definitions

Original Schema

Schema Definitions

This type is an object with the following properties:

description

A description of the transformation performed. *No length restriction*

time

A timestamp for the operation, in UTC. *No length restriction*

software

One or more descriptions of the software that performed the operation.

This node must validate against **any** of the following:

- software-1.0.0
-

  array *No length restriction*

  Items in the array are restricted to the following types:

  software-1.0.0

Original Schema

```
%YAML 1.1
---
$schema: "http://stsci.edu/schemas/yaml-schema/draft-01"
id: "http://stsci.edu/schemas/asdf/core/history_entry-1.0.0"
title: |
  An entry in the file history.
description: |
  A record of an operation that has been performed
  upon a file.

tag: "tag:stsci.edu:asdf/core/history_entry-1.0.0"
type: object
properties:
  description:
    description: |
      A description of the transformation performed.
    type: string

  time:
```

(continues on next page)

(continued from previous page)

```
    description: |
      A timestamp for the operation, in UTC.
    type: string
    format: date-time

  software:
    description: |
      One or more descriptions of the software that performed the
      operation.
    anyOf:
      - $ref: "software-1.0.0"
      - type: array
        items:
          $ref: "software-1.0.0"

required: [description]
additionalProperties: true
...
```

6.1.9 core/extension_metadata-1.0.0

Metadata about specific ASDF extensions that were used to create this file.

Description

Metadata about specific ASDF extensions that were used to create this file.

Outline

Schema Definitions

Original Schema

Schema Definitions

This type is an object with the following properties:

extension_class

The fully-specified name of the extension class. *No length restriction*

package

The name and version of the package that contains the extension.

Original Schema

```
%YAML 1.1
---
$schema: "http://stsci.edu/schemas/yaml-schema/draft-01"
id: "http://stsci.edu/schemas/asdf/core/extension_metadata-1.0.0"
title: |
  Metadata about specific ASDF extensions that were used to create this file.
description: |
  Metadata about specific ASDF extensions that were used to create this file.

tag: "tag:stsci.edu:asdf/core/extension_metadata-1.0.0"
```

(continues on next page)

(continued from previous page)

```
type: object
properties:
  extension_class:
    description: |
      The fully-specified name of the extension class.
    type: string

  package:
    description: |
      The name and version of the package that contains the extension.
    $ref: "software-1.0.0"

required: [extension_class]
...
```

6.1.10 core/integer-1.0.0

Arbitrary precision integer value.

Description

Represents an arbitrarily large integer value.

Outline

Schema Definitions

Examples

Original Schema

Schema Definitions

This type is an object with the following properties:

words

An array of unsigned 32-bit words representing the integer value, stored as little endian (i.e. the first word of the array represents the least significant bits of the integer value).

sign

String indicating whether the integer value is positive or negative. *No length restriction*

Must match the following pattern:

```
^[+-]$
```

string

Optional string representation of the integer value. This field is only intended to improve readability for humans, and therefore no assumptions about format should be made by ASDF readers. *No length restriction*

Examples

An integer value that is stored using an internal array:

```
!core/integer-1.0.0
  sign: +
  string: '11939427705995611438569184383330'
  words: !core/ndarray-1.0.0
    source: 0
    datatype: uint32
    byteorder: little
    shape: [4]
```

The same integer value is stored using an inline array:

```
!core/integer-1.0.0
  sign: +
  string: '11939427705995611438569184383330'
  words: !core/ndarray-1.0.0
    data: [1103110586, 1590521629, 299257845, 15]
    datatype: uint32
    shape: [4]
```

Original Schema

```
%YAML 1.1
---
$schema: "http://stsci.edu/schemas/yaml-schema/draft-01"
id: "http://stsci.edu/schemas/asdf/core/integer-1.0.0"
title: Arbitrary precision integer value.
description: |
  Represents an arbitrarily large integer value.

examples:
  -
    - An integer value that is stored using an internal array
    - |
        !core/integer-1.0.0
          sign: +
          string: '11939427705995611438569184383330'
          words: !core/ndarray-1.0.0
            source: 0
            datatype: uint32
            byteorder: little
            shape: [4]

  -
    - The same integer value is stored using an inline array
    - |
        !core/integer-1.0.0
          sign: +
          string: '11939427705995611438569184383330'
          words: !core/ndarray-1.0.0
            data: [1103110586, 1590521629, 299257845, 15]
            datatype: uint32
            shape: [4]
```

(continues on next page)

(continued from previous page)

```
tag: "tag:stsci.edu:asdf/core/integer-1.0.0"
type: object
properties:
  words:
    $ref: "ndarray-1.0.0"
    description: |
      An array of unsigned 32-bit words representing the integer value, stored
      as little endian (i.e. the first word of the array represents the least
      significant bits of the integer value).
  sign:
    type: string
    pattern: "^[+-]$"
    description: |
      String indicating whether the integer value is positive or negative.
  string:
    type: string
    description: |
      Optional string representation of the integer value. This field is only
      intended to improve readability for humans, and therefore no assumptions
      about format should be made by ASDF readers.
required: [words, sign]
...
```

6.1.11 core/externalarray-1.0.0

Point to an array-like object in an external file.

Description

Allow referencing of array-like objects in external files. These files can be any type of file and in any absolute or relative location to the asdf file. Loading of these files into arrays is not handled by asdf.

Outline

Schema Definitions

Examples

Original Schema

Schema Definitions

This type is an object with the following properties:

fileuri *No length restriction*

target

This node must validate against **any** of the following:

–

integer

–

string *No length restriction*

datatype *No length restriction*

shape *No length restriction* Items in the array must be **any** of the following types:

–

integer

Minimum value: 0

Examples

Example external reference:

```
!core/externalarray-1.0.0
  datatype: int16
  fileuri: aia.lev1_euv_12s.2017-09-06T120001Z.94.image_lev1.fits
  shape: [4096, 4096]
  target: 1
```

Original Schema

```
%YAML 1.1
---
$schema: "http://stsci.edu/schemas/yaml-schema/draft-01"
id: "http://stsci.edu/schemas/asdf/core/externalarray-1.0.0"
tag: "tag:stsci.edu:asdf/core/externalarray-1.0.0"
title: Point to an array-like object in an external file.
description: |
  Allow referencing of array-like objects in external files. These files can be
  any type of file and in any absolute or relative location to the asdf file.
  Loading of these files into arrays is not handled by asdf.
examples:
  -
    - Example external reference
    - |
      !core/externalarray-1.0.0
        datatype: int16
        fileuri: aia.lev1_euv_12s.2017-09-06T120001Z.94.image_lev1.fits
        shape: [4096, 4096]
        target: 1

type: object
properties:
  fileuri:
    type: string
  target:
    anyOf:
      - type: integer
      - type: string
  datatype:
    type: string
  shape:
    type: array
    items:
      anyOf:
      - type: integer
```

(continues on next page)

(continued from previous page)

```
        minimum: 0

required: [fileuri, target, datatype, shape]
additionalProperties: true
...
```

6.1.12 core/subclass_metadata-1.0.0

Metadata on a serialized subclass of an ASDF-enabled type.

Description

Identifies the specific subclass that was serialized, to enable ASDF readers to correctly deserialize the object.

Outline

Schema Definitions

Original Schema

Schema Definitions

This type is an object with the following properties:

name

The name of the subclass that represents this object when deserialized. *No length restriction*

Original Schema

```
%YAML 1.1
---
$schema: "http://stsci.edu/schemas/yaml-schema/draft-01"
id: "http://stsci.edu/schemas/asdf/core/subclass_metadata-1.0.0"
title: |
  Metadata on a serialized subclass of an ASDF-enabled type.
description: |
  Identifies the specific subclass that was serialized,
  to enable ASDF readers to correctly deserialize the object.

tag: "tag:stsci.edu:asdf/core/subclass_metadata-1.0.0"
type: object
properties:
  name:
    description: |
      The name of the subclass that represents this object
      when deserialized.
    type: string

required: [name]
...
```

6.2 FITS

The `fits` module contains schema that support backward compatibility with FITS. It can safely be ignored by most ASDF implementations.

6.2.1 fits/fits-1.0.0

A FITS file inside of an ASDF file.

Description

This schema is useful for distributing ASDF files that can automatically be converted to FITS files by specifying the exact content of the resulting FITS file.

Not all kinds of data in FITS are directly representable in ASDF. For example, applying an offset and scale to the data using the `BZERO` and `BSCALE` keywords. In these cases, it will not be possible to store the data in the native format from FITS and also be accessible in its proper form in the ASDF file.

Only image and binary table extensions are supported.

Outline

Schema Definitions

Examples

Original Schema

Schema Definitions

array *No length restriction*

Items in the array are restricted to the following types:

This type is an object with the following properties:

header

A list of the keyword/value/comment triples from the header, in the order they appear in the FITS file. *No length restriction*

Items in the array are restricted to the following types:

array

Maximum length: 3

The first 3 items in the list must be the following types:

string

Maximum length: 8

Must match the following pattern:

```
[A-Z0-9]*
```

This node must validate against **any** of the following:

string

Maximum length: 60

number

boolean

string

Maximum length: 60

data

The data part of the HDU.

This node must validate against **any** of the following:

- ../core/ndarray-1.0.0
- ../core/table-1.0.0
-

null

Examples

A simple FITS file with a primary header and two extensions:

```
!fits/fits-1.0.0
    - header:
      - [SIMPLE, true, conforms to FITS standard]
      - [BITPIX, 8, array data type]
      - [NAXIS, 0, number of array dimensions]
      - [EXTEND, true]
      - []
      - ['', Top Level MIRI Metadata]
      - []
      - [DATE, '2013-08-30T10:49:55.070373', The date this file was created (UTC)]
      - [FILENAME, MiriDarkReferenceModel_test.fits, The name of the file]
      - [TELESCOP, JWST, The telescope used to acquire the data]
      - []
      - ['', Information about the observation]
      - []
      - [DATE-OBS, '2013-08-30T10:49:55.000000', The date the observation was made (UTC)]
    - data: !core/ndarray-1.0.0
        datatype: float32
        shape: [2, 3, 3, 4]
        source: 0
        byteorder: big
      header:
      - [XTENSION, IMAGE, Image extension]
      - [BITPIX, -32, array data type]
      - [NAXIS, 4, number of array dimensions]
      - [NAXIS1, 4]
      - [NAXIS2, 3]
      - [NAXIS3, 3]
      - [NAXIS4, 2]
      - [PCOUNT, 0, number of parameters]
```

(continues on next page)

(continued from previous page)

```
      - [GCOUNT, 1, number of groups]
      - [EXTNAME, SCI, extension name]
      - [BUNIT, DN, Units of the data array]
    - data: !core/ndarray-1.0.0
        datatype: float32
        shape: [2, 3, 3, 4]
        source: 1
        byteorder: big
      header:
      - [XTENSION, IMAGE, Image extension]
      - [BITPIX, -32, array data type]
      - [NAXIS, 4, number of array dimensions]
      - [NAXIS1, 4]
      - [NAXIS2, 3]
      - [NAXIS3, 3]
      - [NAXIS4, 2]
      - [PCOUNT, 0, number of parameters]
      - [GCOUNT, 1, number of groups]
      - [EXTNAME, ERR, extension name]
      - [BUNIT, DN, Units of the error array]
```

Original Schema

```
%YAML 1.1
---
$schema: "http://stsci.edu/schemas/yaml-schema/draft-01"
id: "http://stsci.edu/schemas/asdf/fits/fits-1.0.0"
title: >
  A FITS file inside of an ASDF file.
description: |
  This schema is useful for distributing ASDF files that can
  automatically be converted to FITS files by specifying the exact
  content of the resulting FITS file.

  Not all kinds of data in FITS are directly representable in ASDF.
  For example, applying an offset and scale to the data using the
  `BZERO` and `BSCALE` keywords.  In these cases, it will not be
  possible to store the data in the native format from FITS and also
  be accessible in its proper form in the ASDF file.

  Only image and binary table extensions are supported.

examples:
  -
    - A simple FITS file with a primary header and two extensions
    - |
        !fits/fits-1.0.0
            - header:
              - [SIMPLE, true, conforms to FITS standard]
              - [BITPIX, 8, array data type]
              - [NAXIS, 0, number of array dimensions]
              - [EXTEND, true]
```

(continues on next page)

(continued from previous page)

```
              - []
              - ['', Top Level MIRI Metadata]
              - []
              - [DATE, '2013-08-30T10:49:55.070373', The date this file was created (UTC)]
              - [FILENAME, MiriDarkReferenceModel_test.fits, The name of the file]
              - [TELESCOP, JWST, The telescope used to acquire the data]
              - []
              - ['', Information about the observation]
              - []
              - [DATE-OBS, '2013-08-30T10:49:55.000000', The date the observation was made␣
↪(UTC)]
            - data: !core/ndarray-1.0.0
                datatype: float32
                shape: [2, 3, 3, 4]
                source: 0
                byteorder: big
              header:
              - [XTENSION, IMAGE, Image extension]
              - [BITPIX, -32, array data type]
              - [NAXIS, 4, number of array dimensions]
              - [NAXIS1, 4]
              - [NAXIS2, 3]
              - [NAXIS3, 3]
              - [NAXIS4, 2]
              - [PCOUNT, 0, number of parameters]
              - [GCOUNT, 1, number of groups]
              - [EXTNAME, SCI, extension name]
              - [BUNIT, DN, Units of the data array]
            - data: !core/ndarray-1.0.0
                datatype: float32
                shape: [2, 3, 3, 4]
                source: 1
                byteorder: big
              header:
              - [XTENSION, IMAGE, Image extension]
              - [BITPIX, -32, array data type]
              - [NAXIS, 4, number of array dimensions]
              - [NAXIS1, 4]
              - [NAXIS2, 3]
              - [NAXIS3, 3]
              - [NAXIS4, 2]
              - [PCOUNT, 0, number of parameters]
              - [GCOUNT, 1, number of groups]
              - [EXTNAME, ERR, extension name]
              - [BUNIT, DN, Units of the error array]

tag: "tag:stsci.edu:asdf/fits/fits-1.0.0"
type: array
items:
  description: >
    Each item represents a single header/data unit (HDU).
  type: object
```

(continues on next page)

(continued from previous page)

```
  properties:
    header:
      description: >
        A list of the keyword/value/comment triples from the header,
        in the order they appear in the FITS file.
      type: array
      items:
        type: array
        minItems: 0
        maxItems: 3
        items:
          - description: "The keyword."
            type: string
            maxLength: 8
            pattern: "[A-Z0-9]*"
          - description: "The value."
            anyOf:
              - type: string
                maxLength: 60
              - type: number
              - type: boolean
          - description: "The comment."
            type: string
            maxLength: 60
    data:
      description: "The data part of the HDU."
      anyOf:
        - $ref: "../core/ndarray-1.0.0"
        - $ref: "../core/table-1.0.0"
        - type: "null"
      default: null
  required: [header]
  additionalProperties: false
...
```

6.3 Unit

The `unit` module contains schema to support the units of physical quantities.

6.3.1 unit/unit-1.0.0

Physical unit.

Description

This represents a physical unit, in VOUnit syntax, Version 1.0 (http://www.ivoa.net/documents/VOUnits/index.html). Where units are not explicitly tagged, they are assumed to be in VOUnit syntax.

Outline

Schema Definitions

Examples

Original Schema

Schema Definitions

This node must validate against **any** of the following:

- tag:stsci.edu:asdf/unit/unit-1.0.0
- *This node has no type definition (unrestricted)*

Examples

Example unit:

```
!unit/unit-1.0.0 "2.1798721 10-18kg m2 s-2"
```

Original Schema

```
%YAML 1.1
---
$schema: "http://stsci.edu/schemas/yaml-schema/draft-01"
id: "http://stsci.edu/schemas/asdf/unit/unit-1.0.0"
title: Physical unit.
description: >
  This represents a physical unit, in [VOUnit syntax, Version
  1.0](http://www.ivoa.net/documents/VOUnits/index.html).

  Where units are not explicitly tagged, they are assumed to be
  in VOUnit syntax.
examples:
  -
    - Example unit
    - |
      !unit/unit-1.0.0 "2.1798721 10-18kg m2 s-2"

anyOf:
  - tag: "tag:stsci.edu:asdf/unit/unit-1.0.0"
  - {}

type: string
pattern: "[\x00-\x7f]*"
...
```

6.3.2 unit/defunit-1.0.0

Define a new physical unit.

Description

Defines a new unit. It can be used to either:

- Define a new base unit.
- Create a new unit name that is a equivalent to a given unit.

The new unit must be defined before any unit tags that use it.

Outline

Schema Definitions

Original Schema

Schema Definitions

This type is an object with the following properties:

name

The name of the new unit. *No length restriction*

Must match the following pattern:

```
[A-Za-z_][A-Za-z0-9_]+
```

unit

The unit that the new name is equivalent to. It is optional, and if not provided, or `null`, this `defunit` defines a new base unit.

This node must validate against **any** of the following:

– unit-1.0.0

–

null

Original Schema

```yaml
%YAML 1.1
---
$schema: "http://stsci.edu/schemas/yaml-schema/draft-01"
id: "http://stsci.edu/schemas/asdf/unit/defunit-1.0.0"
title: Define a new physical unit.
description: |
  Defines a new unit.  It can be used to either:

  - Define a new base unit.

  - Create a new unit name that is a equivalent to a given unit.

  The new unit must be defined before any unit tags that use it.

tag: "tag:stsci.edu:asdf/unit/defunit-1.0.0"
type: object
properties:
  name:
    description: The name of the new unit.
    type: string
    pattern: "[A-Za-z_][A-Za-z0-9_]+"

  unit:
    description: |
      The unit that the new name is equivalent to.  It is optional,
      and if not provided, or ``null``, this ``defunit`` defines a new
      base unit.
```

(continues on next page)

(continued from previous page)

```
    anyOf:
      - $ref: "unit-1.0.0"
      - type: "null"

required: [name]
...
```

6.3.3 unit/quantity-1.1.0

Represents a Quantity object from astropy

Description

A Quantity object represents a value that has some unit associated with the number.

Outline

Schema Definitions

Examples

Original Schema

Schema Definitions

This type is an object with the following properties:

value

A vector of one or more values

This node must validate against **any** of the following:

–

number

– ../core/ndarray-1.0.0

unit

The unit corresponding to the values

Examples

A quantity consisting of a scalar value and unit:

```
!unit/quantity-1.1.0
  value: 3.14159
  unit: km
```

A quantity consisting of a single value in an array:

```
!unit/quantity-1.1.0
  value: !core/ndarray-1.0.0 [2.71828]
  unit: A
```

A quantity with an array of values:

```
!unit/quantity-1.1.0
  value: !core/ndarray-1.0.0 [1, 2, 3, 4]
  unit: s
```

A quantity with an n-dimensional array of values:

```
!unit/quantity-1.1.0
  value: !core/ndarray-1.0.0
    datatype: float64
    data: [[1, 2, 3],
           [4, 5, 6]]
  unit: pc
```

Original Schema

```
%YAML 1.1
---
$schema: "http://stsci.edu/schemas/yaml-schema/draft-01"
id: "http://stsci.edu/schemas/asdf/unit/quantity-1.1.0"
tag: "tag:stsci.edu:asdf/unit/quantity-1.1.0"

title: >
  Represents a Quantity object from astropy
description: |
  A Quantity object represents a value that has some unit
  associated with the number.

examples:
  -
    - A quantity consisting of a scalar value and unit
    - |
        !unit/quantity-1.1.0
          value: 3.14159
          unit: km

  -
    - A quantity consisting of a single value in an array
    - |
        !unit/quantity-1.1.0
          value: !core/ndarray-1.0.0 [2.71828]
          unit: A

  -
    - A quantity with an array of values
    - |
        !unit/quantity-1.1.0
          value: !core/ndarray-1.0.0 [1, 2, 3, 4]
          unit: s

  -
    - A quantity with an n-dimensional array of values
    - |
        !unit/quantity-1.1.0
```

(continues on next page)

(continued from previous page)

```
          value: !core/ndarray-1.0.0
            datatype: float64
            data: [[1, 2, 3],
                   [4, 5, 6]]
          unit: pc


type: object
properties:
  value:
    description: |
      A vector of one or more values
    anyOf:
      - type: number
      - $ref: "../core/ndarray-1.0.0"
  unit:
    description: |
      The unit corresponding to the values
    $ref: unit-1.0.0
required: [value, unit]
...
```

6.4 Time

The `time` module contains schema to support representing instances in time and time deltas.

6.4.1 time/time-1.1.0

Represents an instance in time.

Description

A "time" is a single instant in time. It may explicitly specify the way time is represented (the "format") and the "scale" which specifies the offset and scaling relation of the unit of time.

Specific emphasis is placed on supporting time scales (e.g. UTC, TAI, UT1, TDB) and time representations (e.g. JD, MJD, ISO 8601) that are used in astronomy and required to calculate, e.g., sidereal times and barycentric corrections.

Times may be represented as one of the following:

- an object, with explicit `value`, and optional `format`, `scale` and `location`.
- a string, in which case the format is guessed from across the unambiguous options (`iso`, `byear`, `jyear`, `yday`), and the scale is hardcoded to `UTC`.

In either case, a single time tag may be used to represent an n-dimensional array of times, using either an `ndarray` tag or inline as (possibly nested) YAML lists. If YAML lists, the same format must be used for all time values.

The precision of the numeric formats should only be assumed to be as good as an IEEE-754 double precision (float64) value. If higher-precision is required, the `iso` or `yday` format should be used.

Outline

Schema Definitions

Examples

Internal Definitions

Original Schema

Schema Definitions

This node must validate against **any** of the following:

- #/definitions/string_formats
- #/definitions/array_of_strings
- ../core/ndarray-1.0.0#/anyOf/1
-

This type is an object with the following properties:

value

The value(s) of the time.

This node must validate against **any** of the following:

#/definitions/string_formats

#/definitions/array_of_strings

../core/ndarray-1.0.0

number

format

The format of the time.

If not provided, the the format should be guessed from the string from among the following unambiguous options: `iso`, `byear`, `jyear` and `yday`.

The supported formats are:

`iso`: ISO 8601 compliant date-time format `YYYY-MM-DDTHH:MM:SS.sss...`. For example, `2000-01-01 00:00:00.000` is midnight on January 1, #. The `T` separating the date from the time section is optional.

`yday`: Year, day-of-year and time as `YYYY:DOY:HH:MM:SS.sss...`. The day-of-year (DOY) goes from 001 to 365 (366 in leap years). For example, `2000:001:00:00:00.000` is midnight on January 1, 2000.

`byear`: Besselian Epoch year, eg. `B1950.0`. The `B` is optional if the `byear` format is explicitly specified.

`jyear`: Julian Epoch year, eg. `J2000.0`. The `J` is optional if the `jyear` format is explicitly specified.

`decimalyear`: Time as a decimal year, with integer values corresponding to midnight of the first day of each year. For example 2000.5 corresponds to the ISO time `2000-07-02 00:00:00`.

`jd`: Julian Date time format. This represents the number of days since the beginning of the Julian Period. For example, 2451544.5 in `jd` is midnight on January 1, 2000.

`mjd`: Modified Julian Date time format. This represents the number of days since midnight on November 17, 1858. For example, 51544.0 in MJD is midnight on January 1, 2000.

`gps`: GPS time: seconds from 1980-01-06 00:00:00 UTC For example, 630720013.0 is midnight on January 1, 2000.

`unix`: Unix time: seconds from 1970-01-01 00:00:00 UTC. For example, 946684800.0 in Unix time is midnight on January 1, 2000. [TODO: Astropy's definition of UNIX time doesn't match POSIX's here. What should we do for the purposes of ASDF?]

This node has no type definition (unrestricted)

scale

The time scale (or time standard) is a specification for measuring time: either the rate at which time passes; or points in time; or both. See also [3] and [4].

These scales are defined in detail in SOFA Time Scale and Calendar Tools (http://www.iausofa.org/sofa_ts_c.pdf).

The supported time scales are:

`utc`: Coordinated Universal Time (UTC). This is the default time scale, except for `gps`, `unix`.

`tai`: International Atomic Time (TAI).

`tcb`: Barycentric Coordinate Time (TCB).

`tcg`: Geocentric Coordinate Time (TCG).

`tdb`: Barycentric Dynamical Time (TDB).

`tt`: Terrestrial Time (TT).

`ut1`: Universal Time (UT1).

This node has no type definition (unrestricted)

location

Specifies the observer location for scales that are sensitive to observer location, currently only `tdb`. May be specified either with geocentric coordinates (X, Y, Z) with an optional unit or geodetic coordinates:

`long`: longitude in degrees

`lat`: in degrees

`h`: optional height

This type is an object with the following properties:

x

y

z

Examples

Example ISO time:

```
!time/time-1.1.0 "2000-12-31T13:05:27.737"
```

Example year, day-of-year and time format time:

```
!time/time-1.1.0 "2001:003:04:05:06.789"
```

Example Besselian Epoch time:

```
!time/time-1.1.0 B2000.0
```

Example Besselian Epoch time, equivalent to above:

```
!time/time-1.1.0
  value: 2000.0
  format: byear
```

Example list of times:

```
!time/time-1.1.0
  ["2000-12-31T13:05:27.737", "2000-12-31T13:06:38.444"]
```

Example of an array of times:

```
!time/time-1.1.0
  value: !core/ndarray-1.0.0
    data: [2000, 2001]
    datatype: float64
  format: jyear
```

Example with a location:

```
!time/time-1.1.0
  value: 2000.0
  format: jyear
  scale: tdb
  location:
    x: !unit/quantity-1.1.0
      value: 6378100
      unit: !unit/unit-1.0.0 m
    y: !unit/quantity-1.1.0
      value: 0
      unit: !unit/unit-1.0.0 m
    z: !unit/quantity-1.1.0
      value: 0
      unit: !unit/unit-1.0.0 m
```

Internal Definitions

iso_time *No length restriction*

Must match the following pattern:

```
[0-9]{4}-(0[1-9])|(1[0-2])-(0[1-9])|([1-2][0-9])|(3[0-1])[T ]([0-1][0-9])|(2[0-4]):[0-5][0-
↪9]:[0-5][0-9](.[0-9]+)?
```

byear *No length restriction*

Must match the following pattern:

```
B[0-9]+(.[0-9]+)?
```

jyear *No length restriction*

Must match the following pattern:

```
J[0-9]+(.[0-9]+)?
```

yday *No length restriction*

Must match the following pattern:

```
[0-9]{4}:(00[1-9])|(0[1-9][0-9])|([1-2][0-9][0-9])|(3[0-5][0-9])|(36[0-5]):([0-1][0-9])|([0-
↪1][0-9])|(2[0-4]):[0-5][0-9]:[0-5][0-9](.[0-9]+)?
```

string_formats

This node must validate against **any** of the following:

- #/definitions/iso_time
- #/definitions/byear
- #/definitions/jyear
- #/definitions/yday

array_of_strings *No length restriction* Items in the array must be **any** of the following types:

- #/definitions/array_of_strings
- #/definitions/string_formats

Original Schema

```
%YAML 1.1
---
$schema: "http://stsci.edu/schemas/yaml-schema/draft-01"
id: "http://stsci.edu/schemas/asdf/time/time-1.1.0"
tag: "tag:stsci.edu:asdf/time/time-1.1.0"
title: Represents an instance in time.
description: |
  A "time" is a single instant in time.  It may explicitly specify the
  way time is represented (the "format") and the "scale" which
  specifies the offset and scaling relation of the unit of time.

  Specific emphasis is placed on supporting time scales (e.g. UTC,
  TAI, UT1, TDB) and time representations (e.g. JD, MJD, ISO 8601)
  that are used in astronomy and required to calculate, e.g., sidereal
  times and barycentric corrections.

  Times may be represented as one of the following:

  - an object, with explicit `value`, and optional `format`, `scale`
    and `location`.

  - a string, in which case the format is guessed from across
    the unambiguous options (`iso`, `byear`, `jyear`, `yday`), and the
    scale is hardcoded to `UTC`.

  In either case, a single time tag may be used to represent an
  n-dimensional array of times, using either an `ndarray` tag or
  inline as (possibly nested) YAML lists.  If YAML lists, the same
  format must be used for all time values.
```

(continues on next page)

(continued from previous page)

```
  The precision of the numeric formats should only be assumed to be as
  good as an IEEE-754 double precision (float64) value.  If
  higher-precision is required, the `iso` or `yday` format should be
  used.

examples:
  -
    - Example ISO time
    - |
        !time/time-1.1.0 "2000-12-31T13:05:27.737"

  -
    - Example year, day-of-year and time format time
    - |
        !time/time-1.1.0 "2001:003:04:05:06.789"

  -
    - Example Besselian Epoch time
    - |
        !time/time-1.1.0 B2000.0

  -
    - Example Besselian Epoch time, equivalent to above
    - |
        !time/time-1.1.0
          value: 2000.0
          format: byear

  -
    - Example list of times
    - |
        !time/time-1.1.0
          ["2000-12-31T13:05:27.737", "2000-12-31T13:06:38.444"]

  -
    - Example of an array of times
    - |
        !time/time-1.1.0
          value: !core/ndarray-1.0.0
            data: [2000, 2001]
            datatype: float64
          format: jyear

  -
    - Example with a location
    - |
        !time/time-1.1.0
          value: 2000.0
          format: jyear
          scale: tdb
          location:
```

(continues on next page)

(continued from previous page)

```
            x: !unit/quantity-1.1.0
              value: 6378100
              unit: !unit/unit-1.0.0 m
            y: !unit/quantity-1.1.0
              value: 0
              unit: !unit/unit-1.0.0 m
            z: !unit/quantity-1.1.0
              value: 0
              unit: !unit/unit-1.0.0 m

definitions:
  iso_time:
    type: string
    pattern: "[0-9]{4}-(0[1-9])|(1[0-2])-(0[1-9])|([1-2][0-9])|(3[0-1])[T ]([0-1][0-9])|(2[0-
↪4]):[0-5][0-9]:[0-5][0-9](.[0-9]+)?"

  byear:
    type: string
    pattern: "B[0-9]+(.[0-9]+)?"

  jyear:
    type: string
    pattern: "J[0-9]+(.[0-9]+)?"

  yday:
    type: string
    pattern: "[0-9]{4}:(00[1-9])|(0[1-9][0-9])|([1-2][0-9][0-9])|(3[0-5][0-9])|(36[0-5]):([0-
↪1][0-9])|([0-1][0-9])|(2[0-4]):[0-5][0-9]:[0-5][0-9](.[0-9]+)?"

  string_formats:
    anyOf:
      - $ref: "#/definitions/iso_time"
      - $ref: "#/definitions/byear"
      - $ref: "#/definitions/jyear"
      - $ref: "#/definitions/yday"

  array_of_strings:
    type: array
    items:
      anyOf:
        - $ref: "#/definitions/array_of_strings"
        - $ref: "#/definitions/string_formats"

anyOf:
  - $ref: "#/definitions/string_formats"

  - $ref: "#/definitions/array_of_strings"

  - $ref: "../core/ndarray-1.0.0#/anyOf/1"

  - type: object
    properties:
```

(continues on next page)

(continued from previous page)

```
    value:
      description: |
        The value(s) of the time.

      anyOf:
        - $ref: "#/definitions/string_formats"
        - $ref: "#/definitions/array_of_strings"
        - $ref: "../core/ndarray-1.0.0"
        - type: number

    format:
      description: |
        The format of the time.

        If not provided, the the format should be guessed from the
        string from among the following unambiguous options:
        `iso`, `byear`, `jyear` and `yday`.

        The supported formats are:

        - `iso`: ISO 8601 compliant date-time format
          `YYYY-MM-DDTHH:MM:SS.sss...`.  For example,
          `2000-01-01 00:00:00.000` is midnight on January 1,
          2000.  The `T` separating the date from the time
          section is optional.

        - `yday`: Year, day-of-year and time as
          `YYYY:DOY:HH:MM:SS.sss...`. The day-of-year (DOY) goes
          from 001 to 365 (366 in leap years). For example,
          `2000:001:00:00:00.000` is midnight on January 1,
          2000.

        - `byear`: Besselian Epoch year, eg. `B1950.0`.  The `B`
          is optional if the `byear` format is explicitly
          specified.

        - `jyear`: Julian Epoch year, eg. `J2000.0`.  The `J` is
          optional if the `jyear` format is explicitly
          specified.

        - `decimalyear`: Time as a decimal year, with integer
          values corresponding to midnight of the first day of
          each year. For example 2000.5 corresponds to the ISO
          time `2000-07-02 00:00:00`.

        - `jd`: Julian Date time format. This represents the
          number of days since the beginning of the Julian
          Period. For example, 2451544.5 in `jd` is midnight on
          January 1, 2000.

        - `mjd`: Modified Julian Date time format. This
          represents the number of days since midnight on
```

(continues on next page)

(continued from previous page)

```
          November 17, 1858. For example, 51544.0 in MJD is
          midnight on January 1, 2000.

        - `gps`: GPS time: seconds from 1980-01-06 00:00:00 UTC
          For example, 630720013.0 is midnight on January 1,
          2000.

        - `unix`: Unix time: seconds from 1970-01-01 00:00:00
          UTC. For example, 946684800.0 in Unix time is midnight
          on January 1, 2000.  [TODO: Astropy's definition of
          UNIX time doesn't match POSIX's here.  What should we
          do for the purposes of ASDF?]

      enum:
        - iso
        - yday
        - byear
        - jyear
        - decimalyear
        - jd
        - mjd
        - gps
        - unix
        - cxcsec

    scale:
      description: |
        The time scale (or time standard) is a specification for
        measuring time: either the rate at which time passes; or
        points in time; or both. See also [3] and [4].

        These scales are defined in detail in [SOFA Time Scale and
        Calendar Tools](http://www.iausofa.org/sofa_ts_c.pdf).

        The supported time scales are:

        - `utc`: Coordinated Universal Time (UTC).  This is the
          default time scale, except for `gps`, `unix`.

        - `tai`: International Atomic Time (TAI).

        - `tcb`: Barycentric Coordinate Time (TCB).

        - `tcg`: Geocentric Coordinate Time (TCG).

        - `tdb`: Barycentric Dynamical Time (TDB).

        - `tt`: Terrestrial Time (TT).

        - `ut1`: Universal Time (UT1).

      enum:
```

(continues on next page)

(continued from previous page)

```
          - utc
          - tai
          - tcb
          - tcg
          - tdb
          - tt
          - ut1

      location:
        description: |
          Specifies the observer location for scales that are
          sensitive to observer location, currently only `tdb`.  May
          be specified either with geocentric coordinates (X, Y, Z)
          with an optional unit or geodetic coordinates:
            - `long`: longitude in degrees
            - `lat`: in degrees
            - `h`: optional height

        type: object
        properties:
          x:
            $ref: "../unit/quantity-1.1.0"
          y:
            $ref: "../unit/quantity-1.1.0"
          z:
            $ref: "../unit/quantity-1.1.0"
        required: [x, y, z]

    required: [value]
...
```

The ASDF Standard also defines two meta-schemas that are used for validating the ASDF schemas themselves. These schemas are useful references when creating custom schemas (see *Designing a new tag and schema* (page 23)).

6.5 YAML Schema

YAML Schema (page 81) is a small extension to JSON Schema Draft 4 (http://json-schema.org/draft-04/json-schema-validation.html) that adds some features specific to YAML. Understanding JSON Schema (http://spacetelescope.github.io/understanding-json-schema/) provides a good resource for understanding how to use JSON Schema, and further resources are available at json-schema.org (http://json-schema.org). A working understanding of JSON Schema is assumed for this section, which only describes what makes YAML Schema different from JSON Schema.

Writing a new schema is described in *Designing a new tag and schema* (page 23).

Note: The YAML Schema currently does not require either the `id` or `tag` keywords. The `id` keyword is not included in the YAML Schema since it is actually inherited from the base JSON Schema standard. However, it may become mandatory in a future version of the YAML Standard. The `tag` keyword may also eventually become mandatory, although the motivation for this is somewhat weaker.

YAML Schema

Description

A metaschema extending JSON Schema's metaschema to add support for some YAML-specific constructions.

Outline

Schema Definitions

Internal Definitions

Original Schema

Schema Definitions

This node must validate against **all** of the following:

- http://json-schema.org/draft-04/schema
-

This type is an object with the following properties:

tag

A fully-qualified YAML tag name that should be associated with the object type returned by the YAML parser; for example, the object must be an instance of the class registered with the parser to create instances of objects with this tag. Implementation of this validator is optional and depends on details of the YAML parser.

Minimum length: 6

propertyOrder

Specifies the default order of the properties when writing out. Any keys not listed in **propertyOrder** will be in arbitrary order at the end. This field applies only to nodes with **object** type. *No length restriction*

Items in the array are restricted to the following types:

string *No length restriction*

flowStyle

Specifies the default serialization style to use for an array or object. YAML supports multiple styles for arrays/sequences and objects/maps, called "block style" and "flow style". For example:

```
Block style: !!map
  Clark : Evans
  Ingy  : döt Net
  Oren  : Ben-Kiki

Flow style: !!map { Clark: Evans, Ingy: döt Net, Oren: Ben-Kiki }
```

This property gives a hint to the tool outputting the YAML which style to use. If not provided, the library is free to use whatever heuristics it wishes to determine the output style. This property does not enforce any particular style on YAML being parsed. *No length restriction*

Only the following values are valid for this node:

block

flow

style

Specifies the default serialization style to use for a string. YAML supports multiple styles for strings:

```
Inline style: "First line\nSecond line"

Literal style: |
  First line
  Second line

Folded style: >
  First
  line

  Second
  line
```

This property gives a hint to the tool outputting the YAML which style to use. If not provided, the library is free to use whatever heuristics it wishes to determine the output style. This property does not enforce any particular style on YAML being parsed. *No length restriction*

Only the following values are valid for this node:

inline

literal

folded

examples

A list of examples to help document the schema. Each pair is a prose description followed by a string containing YAML content. For example:

```
examples:
  -
    - Complex number: 1 real, -1 imaginary
    - "!complex 1-1j"
      type: array
      items:
```

No length restriction

Items in the array are restricted to the following types:

array *No length restriction*

The first 2 items in the list must be the following types:

string *No length restriction* This node must validate against **any** of the following:

.

string *No length restriction*

.

object

additionalItems

This node must validate against **any** of the following:

boolean

#

items

This node must validate against **any** of the following:

#

#/definitions/schemaArray

additionalProperties

This node must validate against **any** of the following:

boolean

#

definitions

object

properties

object

patternProperties

object

dependencies

object

allOf

anyOf

oneOf

not

Internal Definitions

schemaArray

Minimum length: 1

Items in the array are restricted to the following types:

#

Original Schema

```
%YAML 1.1
---
$schema: "http://json-schema.org/draft-04/schema"
id: "http://stsci.edu/schemas/yaml-schema/draft-01"
title:
  YAML Schema
description: |
  A metaschema extending JSON Schema's metaschema to add support for
```

(continues on next page)

(continued from previous page)

```
  some YAML-specific constructions.
allOf:
  - $ref: "http://json-schema.org/draft-04/schema"
  - type: object
    properties:
      tag:
        description: |
          A fully-qualified YAML tag name that should be associated
          with the object type returned by the YAML parser; for
          example, the object must be an instance of the class
          registered with the parser to create instances of objects
          with this tag. Implementation of this validator is optional
          and depends on details of the YAML parser.
        type: string
        minLength: 6

      propertyOrder:
        description: |
          Specifies the default order of the properties when writing
          out. Any keys not listed in **propertyOrder** will be in
          arbitrary order at the end. This field applies only to nodes with
          **object** type.
        type: array
        items:
          type: string

      flowStyle:
        description: |
          Specifies the default serialization style to use for an
          array or object.  YAML supports multiple styles for
          arrays/sequences and objects/maps, called "block style" and
          "flow style".  For example::

            Block style: !!map
              Clark : Evans
              Ingy  : döt Net
              Oren  : Ben-Kiki

            Flow style: !!map { Clark: Evans, Ingy: döt Net, Oren: Ben-Kiki }

          This property gives a hint to the tool outputting the YAML
          which style to use.  If not provided, the library is free to
          use whatever heuristics it wishes to determine the output
          style.  This property does not enforce any particular style
          on YAML being parsed.
        type: string
        enum: [block, flow]

      style:
        description: |
          Specifies the default serialization style to use for a string.
          YAML supports multiple styles for strings:
```

(continues on next page)

(continued from previous page)

```
      ```yaml
        Inline style: "First line\nSecond line"

        Literal style: |
          First line
          Second line

        Folded style: >
          First
          line

          Second
          line
      ```

      This property gives a hint to the tool outputting the YAML
      which style to use. If not provided, the library is free to
      use whatever heuristics it wishes to determine the output
      style. This property does not enforce any particular style
      on YAML being parsed.
    type: string
    enum: [inline, literal, folded]

  examples:
    description: |
      A list of examples to help document the schema. Each pair
      is a prose description followed by a string containing YAML
      content. For example:

      ```yaml
        examples:
          -
            - Complex number: 1 real, -1 imaginary
            - "!complex 1-1j"
              type: array
              items:
      ```

    type: array
    items:
      type: array
      items:
        - type: string
        - anyOf:
          - type: string
          - type: object

  # Redefine JSON schema validators in terms of this document so that
  # we can check nested objects:
  additionalItems:
    anyOf:
```

(continues on next page)

(continued from previous page)

```
          - type: boolean
          - $ref: "#"
      items:
        anyOf:
          - $ref: "#"
          - $ref: "#/definitions/schemaArray"
      additionalProperties:
        anyOf:
          - type: boolean
          - $ref: "#"
      definitions:
        type: object
        additionalProperties:
          $ref: "#"
      properties:
        type: object
        additionalProperties:
          $ref: "#"
      patternProperties:
        type: object
        additionalProperties:
          $ref: "#"
      dependencies:
        type: object
        additionalProperties:
          anyOf:
            - $ref: "#"
            - $ref: "http://json-schema.org/draft-04/schema#definitions/stringArray"
      allOf:
        $ref: "#/definitions/schemaArray"
      anyOf:
        $ref: "#/definitions/schemaArray"
      oneOf:
        $ref: "#/definitions/schemaArray"
      not:
        $ref: "#"

definitions:
  schemaArray:
    type: array
    minItems: 1
    items:
      $ref: "#"
...
```

6.6 asdf-schema-1.0.0

ASDF Schema

Description

Extending YAML Schema and JSON Schema to add support for some ASDF-specific checks, related to *ndarrays* (page 34).

Outline

Schema Definitions

Internal Definitions

Original Schema

Schema Definitions

This node must validate against **all** of the following:

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

  This type is an object with the following properties:

  max_ndim

  Specifies that the corresponding **ndarray** is at most the given number of dimensions. If the array has fewer dimensions, it should be logically treated as if it were "broadcast" to the expected dimensions by adding 1's to the front of the shape list.

  Minimum value: 0

  ndim

  Specifies that the matching **ndarray** is exactly the given number of dimensions.

  Minimum value: 0

  datatype

  Specifies the datatype of the **ndarray**.

  By default, an array is considered "matching" if the array can be cast to the given datatype without data loss. For exact datatype matching, set `exact_datatype` to `true`.

  This node must validate against **all** of the following:

  http://stsci.edu/schemas/asdf/core/ndarray-1.0.0#/definitions/datatype

  exact_datatype

  If `true`, the datatype must match exactly.

  Default value:

```
False
```

  additionalItems

  This node must validate against **any** of the following:

  boolean

#

items

This node must validate against **any** of the following:

#

#/definitions/schemaArray

additionalProperties

This node must validate against **any** of the following:

boolean

#

definitions

object

properties

object

patternProperties

object

dependencies

object

allOf

anyOf

oneOf

not

Internal Definitions

schemaArray

Minimum length: 1

Items in the array are restricted to the following types:

#

Original Schema

```
%YAML 1.1
---
$schema: "http://json-schema.org/draft-04/schema"
id: "http://stsci.edu/schemas/asdf/asdf-schema-1.0.0"
title:
  ASDF Schema
description: |
  Extending YAML Schema and JSON Schema to add support for some ASDF-specific
  checks, related to [ndarrays](ref:core/ndarray-1.0.0).
allOf:
  - $ref: "http://stsci.edu/schemas/yaml-schema/draft-01"
```

(continues on next page)

(continued from previous page)

```
  - type: object
    properties:
      max_ndim:
        description: |
          Specifies that the corresponding **ndarray** is at most the
          given number of dimensions.  If the array has fewer
          dimensions, it should be logically treated as if it were
          "broadcast" to the expected dimensions by adding 1's to the
          front of the shape list.
        type: integer
        minimum: 0

      ndim:
        description: |
          Specifies that the matching **ndarray** is exactly the given
          number of dimensions.
        type: integer
        minimum: 0

      datatype:
        description: |
          Specifies the datatype of the **ndarray**.

          By default, an array is considered "matching" if the array
          can be cast to the given datatype without data loss.  For
          exact datatype matching, set `exact_datatype` to `true`.
        allOf:
          - $ref: "http://stsci.edu/schemas/asdf/core/ndarray-1.0.0#/definitions/datatype"

      exact_datatype:
        description: |
          If `true`, the datatype must match exactly.
        type: boolean
        default: false

      # Redefine JSON schema validators in terms of this document so that
      # we can check nested objects:
      additionalItems:
        anyOf:
          - type: boolean
          - $ref: "#"
      items:
        anyOf:
          - $ref: "#"
          - $ref: "#/definitions/schemaArray"
      additionalProperties:
        anyOf:
          - type: boolean
          - $ref: "#"
      definitions:
        type: object
        additionalProperties:
```

(continues on next page)

(continued from previous page)

```
          $ref: "#"
      properties:
        type: object
        additionalProperties:
          $ref: "#"
      patternProperties:
        type: object
        additionalProperties:
          $ref: "#"
      dependencies:
        type: object
        additionalProperties:
          anyOf:
            - $ref: "#"
            - $ref: "http://json-schema.org/draft-04/schema#definitions/stringArray"
      allOf:
        $ref: "#/definitions/schemaArray"
      anyOf:
        $ref: "#/definitions/schemaArray"
      oneOf:
        $ref: "#/definitions/schemaArray"
      not:
        $ref: "#"

definitions:
  schemaArray:
    type: array
    minItems: 1
    items:
      $ref: "#"
...
```

The following graph shows the dependencies between modules:

CHAPTER 7

KNOWN LIMITS

The following is a catalogue of known limits in ASDF 1.6.0.

7.1 Tree

While there is no hard limit on the size of the Tree, in most practical implementations it will need to be read entirely into main memory in order to interpret it, particularly to support forward references. This imposes a practical limit on its size relative to the system memory on the machine. It is not recommended to store large data sets in the tree directly, instead it should reference blocks.

7.2 Literal integer values in the Tree

For practical reasons, integer literals in the Tree must be at most 64-bits within the int64 range. In other words, number must be no greater than 9,223,372,036,854,775,807 or no less than -9,223,372,036,854,775,806.

As of version **1.3.0** of the standard, arbitrary precision integers are supported using *integer* (page 58). Like all tags, use of this type requires library support.

7.3 Blocks

The maximum size of a block header is 65536 bytes.

Since the size of the block is stored in a 64-bit unsigned integer, the largest possible block size is around 18 exabytes. It is likely that other limitations on file size, such as an operating system's filesystem limitations, will be met long before that.

CHAPTER 8

CHANGES

8.1 Version 1.1.0

- `domain` was removed from transforms and was replaced by `bounding_box`. [#138]

8.2 Version 1.0.0

First pre-release.

CHAPTER 9

APPENDIX A: EMBEDDING ASDF IN FITS

While ASDF is designed to replace all of the existing use cases of FITS, there will still be cases where files need to be produced in FITS. Even then, it would be nice to take advantage of the highly-structured nature of ASDF to store content that can not easily be represented in FITS in a FITS file. This appendix describes a convention for embedding ASDF content in a FITS file.

The content of the ASDF file is placed in the data portion of an extra image extension named `ASDF` (`EXTNAME = 'ASDF'`). (By convention, the datatype is unsigned 8-bit integers (`BITPIX = 8`) and is one-dimensional (`NAXIS = 1`), but this is not strictly necessary.)

Rather than including a copy of the large data arrays in the ASDF extension, the ASDF content may refer to binary data stored in regular FITS extensions elsewhere in the same file. The convention for doing this is to set the `source` property of a *ndarray* (page 34) object to a special string identifier for a FITS reference. These values come in two forms:

- `fits:EXTNAME,EXTVER`: Where `EXTNAME` and `EXTVER` uniquely identify a FITS extension.
- `fits:INDEX`: Where `INDEX` is the zero-based index of a FITS extension.

The `fits:EXTNAME,EXTVER` form is preferred, since it allows for rearranging the FITS extensions in the file without the need to update the content of the `ASDF` extension, and thus such rearrangements could be performed by a non-ASDF-aware FITS library.

Such "FITS references" simply point to the binary content of the data portion of a FITS header/data unit. There is no enforcement that the `datatype` of the ASDF *ndarray* (page 34) matches the `BITPIX` of the FITS extension, or expectation that an explicit conversion would be performed if they don't match. It is up to the writer of the file to keep the ASDF and FITS datatype descriptions in sync.

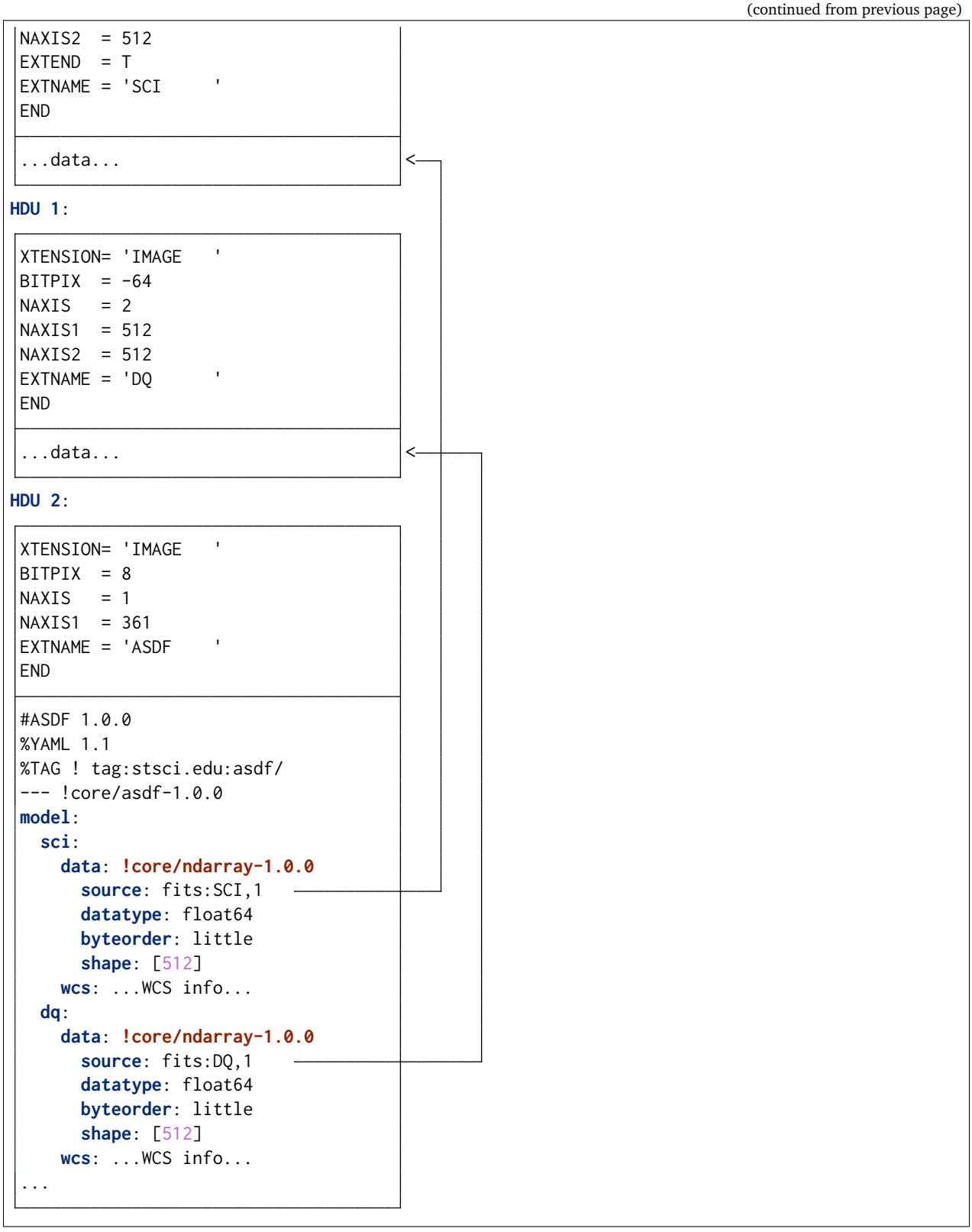
The following is a schematic of an example FITS file with an ASDF extension. The ASDF content references the binary data in two FITS extensions elsewhere in the file.

HDU 0:

```
SIMPLE  = T
BITPIX  = -64
NAXIS   = 2
NAXIS1  = 512
```

(continues on next page)

(continued from previous page)



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/

A paper, ASDF: A new data format for astronomy (https://doi.org/10.1016/j.ascom.2015.06.004) about ASDF has been published in *Astronomy and Computing*:

> Greenfield, P., Droettboom, M., & Bray, E. (2015). ASDF: A new data format for astronomy. *Astronomy and Computing*, 12: 240-251. doi:10.1016/j.ascom.2015.06.004

BIBLIOGRAPHY

[Thomas2015] Thomas, B., Jenness. T. et al. (2015). Learning from FITS: Limitations in use in modern astronomical research. *Astronomy and Computing*, 12: 133-145. doi:10.1016/j.ascom.2015.01.009 (https://doi.org/10.1016/j.ascom.2015.01.009)