Astrodata Programmer Manual

Release 2.1.1

Ricardo Cardenes

April 2020

Contents

Document ID

PIPE-USER-104_AstrodataProgManual

CHAPTER 1

Precedents and Motivation

The Gemini Observatory has produced a number of tools for data processing. Historically this has translated into a number of IRAF[1] packages but the lack of long-term support for IRAF, coupled with the well-known difficulty in creating robust reduction pipelines within the IRAF environment, led to a decision to adopt Python as a programming tool and a new package was born: Gemini Python. Gemini Python provided tools to load and manipulate Gemini-produced multi-extension FITS[2] (MEF) files, along with a pipeline that allowed the construction of reduction recipes. At the center of this package was the AstroData subpackage, which supported the abstraction of the FITS files.

Gemini Python reached version 1.0.1, released during November 2014. In 2015 the Science User Support Department (SUSD) was created at Gemini, which took on the responsibility of maintaining the software reduction tools, and started planning future steps. With improved oversight and time and thought, it became evident that the design of Gemini Python and, specially, of AstroData, made further development a daunting task.

In 2016 a decision was reached to overhaul Gemini Python. While the principles behind AstroData were sound, the coding involved unnecessary layers of abstraction and eschewed features of the Python language in favor of its own implementation. Thus, DRAGONS[3] was born, with a new, simplified (and backward *incompatible*) AstroData v2.0 (which we will refer to simply as AstroData)

This manual documents both the high level design and some implementation details of AstroData, together with an explanation of how to extend the package to work for new environments.

[1] http://iraf.net
[2] The Flexible Image Transport System
[3] The Data Reduction for Astronomy from Gemini Observatory North and South package

CHAPTER 2

General Design

As astronomical instruments have become more complex, there has been an increasing need for bespoke reduction packages and pipelines to deal with the specific needs of each instrument. Despite this complexity, many of the reduction steps can be very similar and the overall effort could be reduced significantly by sharing code. In practice, however, there are often issues regarding the manner in which the data are stored internally. The purpose of AstroData is to provide a uniform interface to the data and metadata, in a manner that is independent both of the specific instrument and the way the data are stored on disk, thereby facilitating this code-sharing. It is *not* a new astronomical data format.

One of the main features of AstroData is the use of *descriptors*, which provide a level of abstraction between the metadata and the code accessing it. Somebody using the AstroData interface who wishes to know the exposure time of a particular astronomical observation represented by the `AstroData` object `ad` can simply write `ad.exposure_time()` without needing to concern themselves about how that value is stored internally, for example, the name of the FITS header keyword. These are discussed further in *Descriptors*.

AstroData also provides a clearer representation of the relationships between different parts of the data produced from a single astronomical observation. Modern astronomical instruments often contain multiple detectors that are read out separately and the multi-extension FITS (MEF) format used by many institutions, including Gemini Observatory, handles the raw data well. In this format, each detector's data and metadata is assigned to its own extension, while there is also a separate extension (the Primary Header Unit, or PHU) containing additional metadata that applies to the entire observation. However, as the data are processed, more data and/or metadata may be added whose relationship is obscured by the limitations of the MEF format. One example is the creation and propagation of information describing the quality and uncertainty of the scientific data: while this was a feature of Gemini IRAF[1], the coding required to implement it was cumbersome and AstroData uses the `astropy.nddata.NDData` class, as discussed in *Data Containers*. This makes the relationship between these data much clearer, and AstroData creates a syntax that makes readily apparent the roles of other data and metadata that may be created during the reduction process.

An `AstroData` object therefore consists of one or more self-contained "extensions" (data and metadata) plus additional data and metadata that is relevant to all the extensions. In many data reduction processes, the same operation will be performed on each extension (e.g., subtracting an overscan region from a CCD frame) and an axiom of AstroData is that iterating over the extensions produces AstroData "slices" which retain knowledge of the top-level data and metadata. Since a slice has one (or more) extensions plus this top-level (meta)data, it too is an `AstroData` object and, specifically, an instance of the same subclass as its parent.

[1] https://www.gemini.edu/sciops/data-and-results/processing-software/description

A final feature of AstroData is the implementation of very high-level metadata. These data, called `tags`, facilitate a key part of the Gemini data reduction system, DRAGONS, by linking the astronomical data to the recipes required to process them. They are explained in detail in *Tags* and the Recipe System Programmers Manual[2].

Note: AstroData and DRAGONS have been developed for the reduction of data from Gemini Observatory, which produces data in the FITS format that is still the most widely-used format for astronomical data. In light of this, and the limited resources in the Science User Support Department, we have only *developed* support for FITS, even though the AstroData format is designed to be independent of the file format. In some cases, this has led to uncertainty and internal disagreement over where precisely to engage in abstraction and, should AstroData support a different file format, we may find alternative solutions that result in small, but possibly significant, changes to the API.

---

[2] PIPE-USER-108_RSProgManual

CHAPTER 3

AstroData and Derivatives

The `astrodata.core.AstroData` class (or simply `astrodata.AstroData`) is the main interface to the package. When opening files or creating new objects, a derivative of this class is returned, as the `AstroData` class is not intended to be used directly. It provides the logic to calculate the *tag set* for an image, which is common to all data products. Aside from that, it lacks any kind of specialized knowledge about the different instruments that produce the FITS files. More importantly, it defines two methods (`info` and `load`) as abstract, meaning that the class cannot be instantiated directly: a derivative must implement those methods in order to be useful. Such derivatives can also implement descriptors, which provide processed metadata in a way that abstracts the user from the raw information (e.g., the keywords in FITS headers).

`AstroData` does define a common interface, though. Much of it consists on implementing semantic behavior (access to components through indices, like a list; arithmetic using standard operators; etc), mostly by implementing standard Python methods:

- Defines a common `__init__` function, that accepts a `DataProvider` as its single argument.
- Implements `__deepcopy__`
- Implements `__iter__` to allow sequential iteration over the main set of components (e.g., FITS science HDUs, but this depends on the DataProvider implementation)
- Implements `__getitem__` to allow data slicing (e.g., `ad[2:4]` returns a new `AstroData` instance that contains only the third and fourth main components)
- Implements `__delitem__` to allow for data removal based on index. It does not define `__setitem__`, though. The basic AstroData series of classes only allows to append new data blocks, not to replace them in one sweeping move
- Implements `__iadd__`, `__isub__`, `__imul__`, `__idiv__`, and their not-in-place versions, based on them.

All of these provide default implementations that rely heavily on the `DataProvider` capabilities. There are a few other methods. For a detailed discussion, please refer to the *API Reference Guide*.

3.1 The `tags` Property

Additionally, and crucial to the package, AstroData offers a `tags` property, that under the hood calculates textual tags that describe the object represented by an instance, and returns a set of strings. Returning a set (as opposed to a list, or other similar structure) is intentional, because it is fast to compare sets, e.g., testing for membership; or calculating intersection, etc., to figure out if a certain dataset belongs to an arbitrary category.

The implementation for the tags property is just a call to `AstroData.__process_tags()`. This function implements the actual logic behind calculating the tag set (described *below*). A derivative class could redefine the algorithm, or build upon it.

3.2 Writing an `AstroData` Derivative

The first step when creating new `AstroData` derivative hierarchy would be to create a new class that knows how to deal with some kind of specific data in a broad sense. DRAGONS provide such a class for FITS files, `astrodata.fits.AstroDataFits`, that can be used as an example for future extensions (e.g., to support the ASDF format).

`AstroDataFits` implements both info and load in ways that are specific to FITS files. It also introduces a number of FITS-specific methods and properties, e.g.:

- The properties `phu` and `hdr`, which return the primary header and a list of headers for the science HDUs, respectively.
- A `write` method, which will write the data back to a FITS file
- A `_matches_data` **static** method, which is very important, involved in guiding for the automatic class choice algorithm during data loading. We'll talk more about this when dealing with *registering our classes*.

It also defines the first few descriptors, which are common to all Gemini data: `instrument`, `object`, and `telescope`, which are good examples of simple descriptors that just map a PHU keyword without applying any conversion.

A typical AstroData programmer will extend this class (`AstroDataFits`), unless introducing support for a different kind of data storage. Any of the classes under the `gemini_instruments` package can be used as examples, but we'll describe the important bits here.

3.2.1 Create a package for it

This is not strictly necessary, but simplifies many things, as we'll see when talking about *registration*. The package layout is up to the designer, so you can decide how to do it. For DRAGONS we've settled on the following recommendation for our internal process (just to keep things familiar):

```
gemini_instruments
  __init__.py
    instrument_name
      __init__.py
        adclass.py
        lookup.py
```

Where `instrument_name` would be the package name (for Gemini we group all our derivative packages under `gemini_instruments`, and we would import `gemini_instruments.gmos`, for example). `__init__.py` and `adclass.py` would be the only required modules under our recommended layout, with `lookup.py` being there just to hold hard-coded values in a module separate from the main logic.

`adclass.py` would contain the declaration of the derivative class, and `__init__.py` will contain any code needed to register our class with the `AstroData` system upon import.

3.2.2 Create your derivative class

This is an excerpt of a typical derivative module:

```
from astrodata import astro_data_tag, astro_data_descriptor, TagSet
from astrodata import AstroDataFits

from . import lookup

class AstroDataInstrument(AstroDataFits):
    __keyword_dict = dict(
            array_name = 'AMPNAME',
            array_section = 'CCDSECT'
        )

    @staticmethod
    def _matches_data(source):
        return source[0].header.get('INSTRUME', '').upper() == 'MYINSTRUMENT'

    @astro_data_tag
    def _tag_instrument(self):
       return TagSet(['MYINSTRUMENT'])

    @astro_data_tag
    def _tag_image(self):
        if self.phu.get('GRATING') == 'MIRROR':
            return TagSet(['IMAGE'])

    @astro_data_tag
    def _tag_dark(self):
        if self.phu.get('OBSTYPE') == 'DARK':
            return TagSet(['DARK'], blocks=['IMAGE', 'SPECT'])

    @astro_data_descriptor
    def array_name(self):
        return self.phu.get(self._keyword_for('array_name'))

    @astro_data_descriptor
    def amp_read_area(self):
        ampname = self.array_name()
        detector_section = self.detector_section()
        return "'{}':{}".format(ampname, detector_section)
```

Note: An actual Gemini Facility Instrument class will derive from `gemini_instruments.AstroDataGemini`, but this is irrelevant for the example.

The class typically relies on functionality declared elsewhere, in some ancestor, e.g., the tag set computation is defined at `AstroData`, and the `_keyword_for` method is defined at `AstroDataFits`.

Some highlights:

- `__keyword_dict`[1] defines one-to-one mappings, assigning a more readable moniker for an HDU header keyword. The idea here is to prevent hard-coding the names of the keywords, in the actual code. While these are typically quite stable and not prone to change, it's better to be safe than sorry, and this can come in useful

[1] Note that the keyword dictionary is a "private" property of the class (due to the double-underscore prefix). Each class can define its own set, which will not be replaced by derivative classes. `_keyword_for` is aware of this and will look up each class up the inheritance chain, in turn, when looking up for keywords.

during instrument development, which is the more likely source of instability. The actual value can be extracted by calling `self._keyword_for('moniker')`.

- `_matches_data` is a static method. It does not have any knowledge about the class itself, and it does not work on an *instance* of the class: it's a member of the class just to make it easier for the AstroData registry to find it. This method is passed some object containing cues of the internal structure and contents of the data. This could be, for example, an instance of `HDUList`, or `DataProvider`. Using these data, `_matches_data` must return a boolean, with `True` meaning "I know how to handle this data".

  Note that `True` **does not mean "I have full knowledge of the data"**. It is acceptable for more than one class to claim compatibility. For a GMOS FITS file, the classes that will return `True` are: `AstroDataFits` (because it is a FITS file that comply with certain minimum requirements), `AstroDataGemini` (the data contains Gemini Facility common metadata), and `AstroDataGmos` (the actual handler!).

  But this does not mean that multiple classes can be valid "final" candidates. If AstroData's automatic class discovery finds more than one class claiming matching with the data, it will start discarding them on the basis of inheritance: any class that appears in the inheritance tree of another one is dropped, because the more specialized one is preferred. If at some point the algorithm cannot find more classes to drop, and there is more than one left in the list, an exception will occur, as AstroData will have no way to choose one over the other.

- A number of "tag methods" have been declared. Their naming is a convention, at the end of the day (the "`_tag_`" prefix, and the related "`_status_`" one, are *just hints* for the programmer): each team should establish a convention that works for them. What is important here is to **decorate** them using `astro_data_tag`, which earmarks the method so that it can be discovered later, and ensures that it returns an appropriate value.

  A tag method will return either a `TagSet` instance (which can be empty), or `None`, which is the same as returning an empty `TagSet`[2].

  All these methods will be executed when looking up for tags, and it's up to the tag set construction algorithm (see *Tags* to figure out the final result. In theory, one **could** provide *just one* big method, but this is feasible only when the logic behind deciding the tag set is simple. The moment that there are a few competing alternatives, with some conditions precluding other branches, one may end up with a rather complicated dozens of lines of logic. Let the algorithm do the heavy work for you: split the tags as needed to keep things simple, with an easy to understand logic.

  Also, keeping the individual (or related) tags in separate methods lets you exploit the inheritance, keeping common ones at a higher level, and redefining them as needed later on, at derived classes.

  Please, refer to `AstroDataGemini`, `AstroDataGmos`, and `AstroDataGnirs` for examples using most of the features.

- The `AstroDataFits.load` method calls the `FitsLoader.load` method, which uses metadata in the FITS headers to determine how the data should be stored in the `AstroData` object. In particular, the `EXTNAME` and `EXTVER` keywords are used to assign individual FITS HDUs, using the same names (`SCI`, `DQ`, and `VAR`) as Gemini-IRAF for the `data`, `mask`, and `variance` planes. A `SCI` HDU *must* exist if there is another HDU with the same `EXTVER`, or else an error will occur.

  If the raw data do not conform to this format, the `AstroDataFits.load` method can be overridden by your class, by having it call the `FitsLoader.load` method with an additional parameter, *extname_parser*, that provides a function to modify the header. This function will be called on each HDU before further processing. As an example, the SOAR Adaptive Module Imager (SAMI) instrument writes raw data as a 4-extension MEF file, with the extensions having `EXTNAME` values `im1`, `im2`, etc. These need to be modified to `SCI`, and an appropriate `EXTVER` keyword added'[3]. This can be done by writing a suitable `load` method for the `AstroDataSami` class:

[2] Notice that the example functions will return only a `TagSet`, if appropriate. This is OK, remember that *every function* in Python returns a value, which will be `None`, implicitly, if you don't specify otherwise.

[3] An `EXTVER` keyword is not formally required as the `FitsLoader.load` method will assign the lowest available integer to a `SCI` header with no `EXTVER` keyword (or if its value is -1). But we wish to be able to identify the original `im1` header by assigning it an `EXTVER` of 1, etc.

```
@classmethod
def load(cls, source):
    def sami_parser(hdu):
        m = re.match('im(\d)', hdu.header.get('EXTNAME', ''))
        if m:
            hdu.header['EXTNAME'] = ('SCI', 'Added by AstroData')
            hdu.header['EXTVER'] = (int(m.group(1)), 'Added by AstroData')

    return cls(FitsLoader(FitsProvider).load(source, extname_parser=sami_parser))
```

- *Descriptors* will make the bulk of the class: again, the name is arbitrary, and it should be descriptive. What *may* be important here is to use `astro_data_descriptor` to decorate them. This is *not required*, because unlike tag methods, descriptors are meant to be called explicitly by the programmer, but they can still be earmarked (using this decorator) to be listed when calling the `descriptors` property. The decorator does not alter the descriptor input or output in any way, so it is always safe to use it, and you probably should, unless there's a good reason against it (e.g., if a descriptor is deprecated and you don't want it to show up in lookups).

  More detailed information can be found in *Descriptors*.

3.2.3 Register your class

Finally, you need to include your class in the **AstroData Registry**. This is an internal structure with a list of all the `AstroData`-derived classes that we want to make available for our programs. Including the classes in this registry is an important step, because a file should be opened using `astrodata.open` or `astrodata.create`, which uses the registry to identify the appropriate class (via the `_matches_data` methods), instead of having the user specify it explicitly.

The version of AstroData prior to DRAGONS had an auto-discovery mechanism, that explored the source tree looking for the relevant classes and other related information. This forced a fixed directory structure (because the code needed to know where to look for files), and gave the names of files and classes semantic meaning (to know *which* files to look into, for example). Aside from the rigidness of the scheme, this introduced all sort of inefficiencies, including an unacceptably high overhead when importing the AstroData package for the first time during execution.

In this new version of AstroData we've introduced a more manageable scheme, that places the discovery responsibility on the programmer. A typical `__init__.py` file on an instrument package will look like this:

```
__all__ = ['AstroDataMyInstrument']

from astrodata import factory
from .adclass import AstroDataMyInstrument

factory.addClass(AstroDataMyInstrument)
```

The call to `factory.addClass` is the one registering the class. This step **needs** to be done **before** the class can be used effectively in the AstroData system. Placing the registration step in the `__init__.py` file is convenient, because importing the package will be enough!

Thus, a script making use of DRAGONS' AstroData to manipulate GMOS data could start like this:

```
import astrodata
from gemini_instruments import gmos

...

ad = astrodata.open(some_file)
```

The first import line is not needed, technically, because the `gmos` package will import it too, anyway, but we'll probably need the `astrodata` package in the namespace anyway, and it's always better to be explicit. Our typical DRAGONS scripts and modules start like this, instead:

```
import astrodata
import gemini_instruments
```

`gemini_instruments` imports all the packages under it, making knowledge about all Gemini instruments available for the script, which is perfect for a multi-instrument pipeline, for example. Loading all the instrument classes is not typically a burden on memory, though, so it's easier for everyone to take the more general approach. It also makes things easier on the end user, because they won't need to know internal details of our packages (like their naming scheme). We suggest this "*cascade import*" scheme for all new source trees, letting the user decide which level of detail they need.

As an additional step, the `__init__.py` file in a package may do extra initialization. For example, for the Gemini modules, one piece of functionality that is shared across instruments is a descriptor that translates a filter's name (say "u" or "FeII") to its central wavelength (e.g., 0.35µm, 1.644µm). As it is a rather common function for us, it is implemented by `AstroDataGemini`. This class **does not know** about its daughter classes, though, meaning that it **cannot know** about the filters offered by their instruments. Instead, we offer a function that can be used to update the filter → wavelength mapping in `gemini_instruments.gemini.lookup` so that it is accessible by the `AstroDataGemini`-level descriptor. So our `gmos.__init__.py` looks like this:

```
__all__ = ['AstroDataGmos']

from astrodata import factory
from ..gemini import addInstrumentFilterWavelengths
from .adclass import AstroDataGmos
from .lookup import filter_wavelengths

factory.addClass(AstroDataGmos)
# Use the generic GMOS name for both GMOS-N and GMOS-S
addInstrumentFilterWavelengths('GMOS', filter_wavelengths)
```

where `addInstrumentfilterWavelengths` is provided by the `gemini` package to perform the update in a controlled way.

We encourage package maintainers and creators to follow such explicit initialization methods, driven by the modules that add functionality themselves, as opposed to active discovery methods on the core code. This favors decoupling between modules, which is generally a good idea.

CHAPTER 4

Data Providers

`AstroData` derivative classes act as a front end. Most of the heavy lifting is actually performed by a `DataProvider` class. There will typically be one data provider per kind of data structure (so far, DRAGONS offers only `astrodata.fits.FitsProvider`), and possibly one data provider **proxy**, used to simplify the handling of data slicing (mapping most of it operations to a regular data provider.)

The data provider acts as a hierarchical data storage. At the **top level**, it contains:

- A sequence of "extensions", representing individual data planes and their associated metadata, likely to represent separate detectors or amplifiers. One can access these extensions by index (e.g., `ad[5]`). Indexing starts at 0, following Python's convention.
- Global objects like masks or tables affecting all the extensions.

Each extension, in turn, is an instance of a Data Container, keeping important metadata (e.g., a FITS HDU's header) and the main data for the extension (e.g., the data for a SCI extension, on Gemini data), along with any other associated data (masks, variance plane, tables, etc).

`astrodata.core.DataProvider` is, again, an abstract class, defining the minimum interface expected from a data provider. This interface is described in greater detail in the *API Reference Guide*, but among other things, one would need to implement:

- `is_settable`: AstroData exposes attributes from its data provider through its own `__getattr__` and `__setattr__`. When trying to set a value for an attribute, `AstroData` will use this method to discover whether the attribute can be modified.
- `append`: a very important method, used to add new top-level components to the provider.
- `__getitem__`: which returns a sliced **view**[1] of the provider itself, meant to work with isolated extensions (examples of such a view are instances of `astrodata.fits.FitsProviderProxy`). The view should behave in almost every way as a normal provider.
- `__len__`: number of science extensions contained in this instance.
- `__iadd__`, `__isub__`, `__imul__`, `__idiv__`; used to perform in-place operations over the data.

[1] For efficiency reasons, and to keep just one version of the data. The method may decide to return a sliced copy instead, but this is a design decision.

- `data`, `uncertainty`, `mask`, `variance`: properties used to access certain common content. These methods generally return lists, with one element per extension.

There are also a number of properties that are not declared as abstract, but still need to be reimplemented if one would want any kind of proper behavior from the class: `exposed` (used to determine if a certain attribute is to be "exposed" to the user through the AstroData class), `is_sliced`, and `is_single`. Of particular interest is this later one: `is_single` is a predicate that should return `True` only if a data provider has been sliced using a single index, e.g.:

```
>>> d1 = provider[:4]
>>> d1.is_sliced, d1.is_single
(True, False)
>>> d2 = provider[3]
>>> d2.is_sliced, d2.is_single
(True, True)
```

This is important for the AstroData interface. When a data provider is being considered a "single" slice, the behavior of many methods change. For example, we mentioned that the `data` property *generally* returns a list. **If the data provider in question is a single slice, then data would return a single (i.e., scalar) element**. This behavior is often seen also in *Descriptors*. Refer always to the to documentation of a method to figure out how they behave. As programmers, you should always include this explicitly in the documentation, even if it's implicit to AstroData.

4.1 Implementation Guidelines

AstroData does not impose any restriction on how to organize the data internally, or how to deal with slicing. On slicing, we chose to use a "proxy" class for `FitsProvider`. So, when sliced (through __getitem__), a `FitsProvider` will return a `FitsProviderProxy`, which is also a descendant of `DataProvider` and reproduces the interface of its "proxied" class.

More importantly, `FitsProviderProxy` keeps an internal mapping of the sliced extensions. So, we may be referring to `sliced[0]` and this would be mapped to, say, `nonsliced[3]`. `FitsProviderProxy`.

Both `FitsProvider` and `FitsProviderProxy` can be studied as an example implementation, but there is no need to follow them: please, evaluate carefully the needs for your design, and feel free to depart from ours. As long as the minimum interface is honored, AstroData will work as intended.

Note also that these classes were subject to heavy changes during development and a future release cycle should see them refactored for clarity and to drop any remnants of interfaces that were deprecated before the initial DRAGONS public release.

As a last comment: remember that `AstroData` exposes its underlying `DataProvider` interface up to a certain point. This can be used to dynamically expose to the user additional attributes, dependent on the underlying technology, or even to the instrument, if needed. This is all fine and encouraged **as long as everything is well documented**, and the user understands that certain parts of the interface may not be available when using different observatory's files[2], for example.

4.2 Registering a Data Provider to be Used with AstroData

Once we have a new data provider class, we need to let AstroData know how to use it, and which class will make use of it. Normally a new data provider will be associated to a new second level AstroData class (ie. a direct descendant to `AstroData`, and a sibling of `AstroDataFits`). This does not have to be always the case, though: if an observatory organizes their FITS files in a way that significantly departs from Gemini's, then creating a separate data provider may be justified, if it makes it easier to deal with the data.

[2] At the time of writing this manual, SOAR has extended DRAGONS for their own use, but they are using the core FITS capabilities as defined by Gemini's implementation.

There are no instances of this as of yet, but we've made a conscious effort during the design phase to make as easy as possible to plug in new providers. Future release of this document will address this topic.

CHAPTER 5

Data Containers

A third, and very important part of the AstroData core package is the data container. We have chosen to extend Astropy's NDData[1] with our own requirements, particularly lazy-loading of data using by opening the FITS files in read-only, memory-mapping mode, and exploiting the windowing capability of PyFITS[2] (using `section`) to reduce our memory requirements, which becomes important when reducing data (e.g., stacking).

We document our container for completeness and for reference, but note that its use is intimately linked to `FitsProvider`. If you're implementing an alternative data provider, you do not need to follow our design.

We'll describe here how we depart from NDData, and how do we integrate the data containers with the rest of the package. Please refer to NDData for the full interface.

Our main data container is `astrodata.nddata.NDAstroData`. Fundamentally, it is a derivative of `astropy.nddata.NDData`, plus a number of mixins to add functionality:

```
class NDAstroData(NDArithmeticMixin, NDSlicingMixin, NDData):
    ...
```

This allows us out of the box to have proper arithmetic with error propagation, and slicing the data with the array syntax.

Our first customization is `NDAstroData.__init__`. It relies mostly on the upstream initialization, but customizes it because our class is initialized with lazy-loaded data wrapped around a custom class (`astrodata.fits.FitsLazyLoadable`) that mimics a PyFITS HDU instance just enough to play along with NDData's initialization code.

`FitsLazyLoadable` is an integral part of our memory-mapping scheme, and among other things it will scale data on the fly, as memory-mapped FITS data can only be read unscaled. Our NDAstroData redefines the properties `data`, `uncertainty`, and `mask`, in two ways:

- To deal with the fact that our class is storing `FitsLazyLoadable` instances, not arrays, as `NDData` would expect. This is to keep data out of memory as long as possible.
- To replace lazy-loaded data with a real in-memory array, under certain conditions (e.g., if the data is modified, as we won't apply the changes to the original file!)

[1] Astropy: N-dimensional datasets

[2] I mention PyFITS because of familiarity and for short, but in reality we're using Astropy's `fits.io` module.

Our obsession with lazy-loading and discarding data is directed to reduce memory fragmentation as much as possible. This is a real problem that can hit applications dealing with large arrays, particularly when using Python. Given the choice to optimize for speed or for memory consumption, we've chosen the latter, which is the more pressing issue.

Another addition of as is the `variance` property as a convenience for the user. Astropy, so far, only provides a standard deviation class for storing uncertainties and the code to propagate errors stored this way already exists. However, our coding elsewhere is greatly simplified if we are able to access and set the variance directly.

Lastly, we've added another new property, `window`, that can be used to explicitly exploit the PyFITS `section` property, to (again) avoid loading unneeded data to memory. This property returns an instance of `NDWindowing` which, when sliced, in turn produces an instance of `NDWindowingAstroData`, itself a proxy of `NDAstroData`. This scheme may seem complex, but it was deemed the easiest and cleanest way to achieve the result that we were looking for.

Note: We expect to make changes to `NDAstroData` in future releases. In particular, we plan to make use of the `wcs` and `unit` attributes provided by the `NDData` class and increase the use of memory-mapping by default. These changes mostly represent increased functionality and we anticipate a high (and possibly full) degree of backward compatibility.

CHAPTER 6

Tags

We described *above* how to generate tags for an AstroData derivative. In this section we'll describe the algorithm that generates the complete tag set out of the individual `TagSet` instances. The algorithm collects all the tags in a list and then decides whether to apply them or not following certain rules, but let's talk about `TagSet` first.

`TagSet` is actually a standard named tuple customized to generate default values (`None`) for its missing members. Its signature is:

```
TagSet(add=None, remove=None, blocked_by=None, blocks=None,
       if_present=None)
```

The most common `TagSet` is an **additive** one: `TagSet(['FOO', 'BAR'])`. If all you need is to add tags, then you're done here. But the real power of our tag generating system is that you can specify some conditions to apply a certain `TagSet`, or put restrictions on others. The different arguments to `TagSet` all expect a list (or some other work the in the following way):

- `add`: if this `TagSet` is selected, then add all these members to the tag set.
- `remove`: if this `TagSet` is selected, then prevent all these members from joining the tag set.
- `blocked_by`: if any of the tags listed in here exist in the tag set, then discard this `TagSet` altogether.
- `blocks`: discard from the list of unprocessed ones any `TagSet` that would add any of the tags listed here.
- `if_present`: process this tag only if all the tags listed in here exist in the tag set at this point.

Note that `blocked_by` and `blocks` look like two sides of the same coin. This is intentional: which one to use is up to the programmer, depending on what will reduce the amount typing and/or make the logic easier (sometimes one wants to block a bunch of other tags from a single one; sometimes one wants a tag to be blocked by a bunch of others). Furthermore, while `blocks` and `blocked_by` prevent the entire `TagSet` from being added if it contains a tag affected by these, `remove` only affects the specific tag.

Now, the algorithm works like this:

1. Collect all the `TagSet` generated by methods in the instance that are decorated using `astro_data_tag`.
2. Then we sort them out:

1. Those that subtract tags from the tag set go first (the ones with non-empty `remove` or `blocks`), allowing them to act early on
2. Those with non-empty `blocked_by` are moved to the end of the list, to ensure that other tags can be generated before them.
3. Those with non-empty `if_present` are moved behind those with `blocked_by`.

3. Now that we've sorted the tags, process them sequentially and for each one:
   1. If they require other tags to be present, make sure that this is the case. If the requirements are not met, drop the tagset. If not...
   2. Figure out if any other tag is blocking the tagset. This will be the case if *any* of the tags to be added is in the "blocked" list, or if any of the tags added by previous tag sets are in the `blocked_by` list of the one being processed. Then...
   3. If all the previous hurdles have been passed, apply the changes declared by this tag (add, remove, and/or block others).

Note that Python's sort algorithm is stable. This means, that if two elements are indistinguishable from the point of view of the sorting algorithm, they are guaranteed to stay in the same relative position. To better understand how this affects our tags, and the algorithm itself, let's follow up with an example taken from real code (the Gemini-generic and GMOS modules):

```
# Simple tagset, with only a constant, additive content
@astro_data_tag
def _tag_instrument(self):
    return TagSet(['GMOS'])

# Simple tagset, also with additive content. This one will
# check if the frame fits the requirements to be classified
# as "GMOS imaging". It returns a value conditionally:
# if this is not imaging, then it will return None, which
# means the algorithm will ignore the value
@astro_data_tag
def _tag_image(self):
    if self.phu.get('GRATING') == 'MIRROR':
        return TagSet(['IMAGE'])

# This is a slightly more complex TagSet (but fairly simple, anyway),
# inherited by all Gemini instruments.
@astro_data_tag
def _type_gcal_lamp(self):
    if self.phu.get('GCALLAMP') == 'IRhigh':
        shut = self.phu.get('GCALSHUT')
        if shut == 'OPEN':
            return TagSet(['GCAL_IR_ON', 'LAMPON'],
                          blocked_by=['PROCESSED'])
        elif shut == 'CLOSED':
            return TagSet(['GCAL_IR_OFF', 'LAMPOFF'],
                          blocked_by=['PROCESSED'])

# This tagset is only active when we detect that the frame is
# a bias. In that case we want to prevent the frame from being
# classified as "imaging" or "spectroscopy", which depend on the
# configuration of the instrument
@astro_data_tag
def _tag_bias(self):
```

(continues on next page)

(continued from previous page)

```
    if self.phu.get('OBSTYPE') == 'BIAS':
        return TagSet(['BIAS', 'CAL'], blocks=['IMAGE', 'SPECT'])
```

These four simple tag methods will serve to illustrate the algorithm. Let's pretend that the requirements for all four of them are somehow met, meaning that we get four `TagSet` instances in our list, in some random order. After step 1 in the algorithm, then, we may have collected the following list:

```
[ TagSet(['GMOS']),
  TagSet(['GCAL_IR_OFF', 'LAMPOFF'], blocked_by=['PROCESSED']),
  TagSet(['BIAS', 'CAL'], blocks=['IMAGE', 'SPECT']),
  TagSet(['IMAGE']) ]
```

The algorithm then proceeds to sort them. First, it will promote the `TagSet` with non-empty `blocks` or `remove`:

```
[ TagSet(['BIAS', 'CAL'], blocks=['IMAGE', 'SPECT']),
  TagSet(['GMOS']),
  TagSet(['GCAL_IR_OFF', 'LAMPOFF'], blocked_by=['PROCESSED']),
  TagSet(['IMAGE']) ]
```

Note that the other three `TagSet` stay in exactly the same order. Now the algorithm will sort the list again, moving the ones with non-empty `blocked_by` to the end:

```
[ TagSet(['BIAS', 'CAL'], blocks=['IMAGE', 'SPECT']),
  TagSet(['GMOS']), TagSet(['IMAGE']),
  TagSet(['GCAL_IR_OFF', 'LAMPOFF'], blocked_by=['PROCESSED']) ]
```

Note that at each step, all the instances (except the ones "being moved") have kept the same position relative to each other -here's where the "stability" of the sorting comes into play,- ensuring that each step does not affect the previous one. Finally, there are no `if_present` in our example, so no more instances are moved around.

Now the algorithm prepares three empty sets (`tags`, `removals`, and `blocked`), and starts iterating over the `TagSet` list.

1. For the first `TagSet` there are no blocks or removals, so we just add its contents to the current sets: `tags = {'BIAS', 'CAL'}`, `blocks = {'IMAGE', 'SPECT'}`.
2. Then comes `TagSet(['GMOS'])`. Again, there are no removals in place, and `GMOS` is not in the list of blocked tags. Thus, we just add it to the current tag set: `tags = {'BIAS', 'CAL', 'GMOS'}`.
3. When processing `TagSet(['IMAGE'])`, the algorithm observes that this `IMAGE` is in the `blocked` set, and stops processing this tag set.
4. Finally, neither `GCAL_IR_OFF` nor `LAMPOFF` are in `blocked`, and `PROCESSED` is not in `tags`, meaning that we can add add this tag set to the final one.

Our result will look something like: `{'BIAS', 'CAL', 'GMOS', 'GCAL_IR_OFF', 'LAMPOFF'}`

CHAPTER 7

Descriptors

Descriptors are just regular methods that translate metadata from the raw storage (e.g., cards from FITS headers) to values useful for the user, potentially doing some processing in between. They exist to:

- Abstract the actual organization of the metadata; e.g. `AstroDataGemini` takes the detector gain from a keyword in the FITS PHU, where `AstroDataNiri` overrides this to provide a hard-coded value.

  More complex implementations also exist. In order to determine the gain of a GMOS observations, `AstroDataGmos` uses the observation date (provided by a descriptor) to select a particular lookup table, and then uses the values of other descriptors to select the correct entry in the table.

- Provide a common interface to a set of instruments. This simplifies user training (no need to learn a different API for each instrument), and facilitates the reuse of code for pipelines, etc.

- Also, since FITS header keywords are limited to 8 characters, for simple keyword → value mappings, they provide a more meaningful and readable name.

Descriptors **should** be decorated using `astrodata.core.astro_data_descriptor`. The only function of this decorator is to ensure that the descriptor is marked as such: it does not alter its input or output in any way. This lets the user to explore the API of an AstroData object via the `descriptors` property.

Descriptors **can** be decorated with `astrodata.core.returns_list` to eliminate the need to code some logic. Some descriptors return single values, while some return lists, one per extension. Typically, the former are descriptors that refer to the entire observation (and, for MEF files, are usually extracted from metadata in the PHU, such as `airmass`), while the latter are descriptors where different extensions might return different values (and typically come from metadata in the individual HDUs, such as `gain`). A list is returned even if there is only one extension in the AstroData object, as this allows code to be written generically to iterate over the `AstroData` object and the descriptor return, without needing to know how many extensions there are. The `returns_list` decorator ensures that the descriptor returns an appropriate object (value or list), using the following rules:

- If the `AstroData` object is not a single slice:
  - If the undecorated descriptor returns a list, an exception is raised if the list is not the same length as the number of extensions.
  - If the undecorated descriptor returns a single value, the decorator will turn it into a list of the correct length by copying this value.

- If the `AstroData` object is a single slice and the undecorated descriptor returns a list, only the first element is returned.

An example of the use of this decorator is the `AstroDataNiri gain` descriptor, which reads the value from a lookup table and simply returns it. A single value is only appropriate if the AstroData object is singly-sliced and the decorator ensures that a list is returned otherwise.

APPENDIX A

API Reference Guide

A.1 Abstract Classes

These classes are the top of their respective hierarchies, and need to be fully implemented before being used. DRAGONS ships with implementations covering the usage of Gemini-style FITS files.

A.1.1 AstroData

class astrodata.core.**AstroData**(*provider*)

Base class for the AstroData software package. It provides an interface to manipulate astronomical data sets.

Parameters **provider** (DataProvider) – The data that will be manipulated through the *AstroData* instance.

add(*oper*)

Alias for __iadd__

subtract(*oper*)

Alias for __isub__

multiply(*oper*)

Alias for __imul__

divide(*oper*)

Alias for __idiv__

__itruediv__(*oper*)

Alias for __idiv__

__add__(*oper*)

Implements the binary arithmetic operation + with *AstroData* as the left operand.

Parameters **oper** (*number or object*) – The operand to be added to this instance. The accepted types depend on the *DataProvider*

Returns

Return type A new *AstroData* instance

__contains__(*attribute*)
Implements the ability to use the *in* operator with an *AstroData* object. It will look up the specified attribute name within the exposed members of the internal *DataProvider* object. Refer to the concrete *DataProvider* implementation's documentation to know what members are exposed.

Parameters **attribute** (*string*) – An attribute name

Returns

Return type A boolean

__deepcopy__(*memo*)
Returns a new instance of this class, initialized with a deep copy of the associted *DataProvider*

Parameters **memo** (*dict*) – See the documentation on *deepcopy* for an explanation on how this works

Returns

Return type A deep copy of this instance

__delattr__(*attribute*)
Implements attribute removal. If *self* represents a single slice, the

__delitem__(*idx*)
Called to implement deletion of *self[idx]*. Supports standard Python syntax (including negative indices).

Parameters **idx** (*integer*) – This index represents the order of the element that you want to remove.

Raises IndexError – If *idx* is out of range

__div__(*oper*)
Implements the binary arithmetic operation / with *AstroData* as the left operand.

Parameters **oper** (*number or object*) – The operand to be added to this instance. The accepted types depend on the *DataProvider*

Returns

Return type A new *AstroData* instance

__getattr__(*attribute*)
Called when an attribute lookup has not found the attribute in the usual places (not an instance attribute, and not in the class tree for *self*).

This is implemented to provide access to objects exposed by the *DataProvider*

Parameters **attribute** (*string*) – The attribute's name

Raises AttributeError – If the attribute could not be found/computed.

__getitem__(*slicing*)
Returns a sliced view of the instance. It supports the standard Python indexing syntax.

Parameters **slice** (int, *slice*) – An integer or an instance of a Python standard *slice* object

Raises

- TypeError – If trying to slice an object when it doesn't make sense (eg. slicing a single slice)
- ValueError – If *slice* does not belong to one of the recognized types
- IndexError – If an index is out of range

Examples

```
>>> single = ad[0]
>>> multiple = ad[:5]
```

__iadd__(*oper*)
Implements the augmented arithmetic assignment +=.

Parameters **oper** (*number or object*) – The operand to be added to this instance. The accepted types depend on the *DataProvider*

Returns

Return type *self*

__idiv__(*oper*)
Implements the augmented arithmetic assignment /=.

Parameters **oper** (*number or other*) – The operand to be added to this instance. The accepted types depend on the *DataProvider*

Returns

Return type *self*

__imul__(*oper*)
Implements the augmented arithmetic assignment *=.

Parameters **oper** (*number or object*) – The operand to be added to this instance. The accepted types depend on the *DataProvider*

Returns

Return type *self*

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

__isub__(*oper*)
Implements the augmented arithmetic assignment -=.

Parameters **oper** (*number or object*) – The operand to be added to this instance. The accepted types depend on the *DataProvider*

Returns

Return type *self*

__len__()
Number of independent extensions stored by the *DataProvider*

Returns

Return type A non-negative integer.

__mul__(*oper*)
Implements the binary arithmetic operation * with *AstroData* as the left operand.

Parameters **oper** (*number or object*) – The operand to be added to this instance. The accepted types depend on the *DataProvider*

Returns

Return type A new *AstroData* instance

__radd__(*oper*)
Implements the binary arithmetic operation + with *AstroData* as the left operand.

Parameters **oper** (*number or object*) – The operand to be added to this instance. The accepted types depend on the *DataProvider*

Returns

Return type A new *AstroData* instance

__rmul__(*oper*)
Implements the binary arithmetic operation * with *AstroData* as the left operand.

Parameters **oper** (*number or object*) – The operand to be added to this instance. The accepted types depend on the *DataProvider*

Returns

Return type A new *AstroData* instance

__setattr__(*attribute*, *value*)
Called when an attribute assignment is attempted, instead of the normal mechanism. This method will check first with the *DataProvider*: if the DP says it will contain this attribute, or that it will accept it for setting, then the value will be stored at the DP level. Otherwise, the regular attribute assignment mechanisme takes over and the value will be store as an instance attribute of *self*.

Parameters

- **attribute** (*string*) – The attribute's name
- **value** (*object*) – The value to be assigned to the attribute

Returns

- If the value is passed to the *DataProvider*, and it is not of an acceptable type,
- a *ValueError* (or other exception) may be rised. Please, check the appropriate
- *documentation for this.*

__sub__(*oper*)
Implements the binary arithmetic operation - with *AstroData* as the left operand.

Parameters **oper** (*number or object*) – The operand to be added to this instance. The accepted types depend on the *DataProvider*

Returns

Return type A new *AstroData* instance

__truediv__(*oper*)
Implements the binary arithmetic operation / with *AstroData* as the left operand.

Parameters **oper** (*number or object*) – The operand to be added to this instance. The accepted types depend on the *DataProvider*

Returns

Return type A new *AstroData* instance

__weakref__
list of weak references to the object (if defined)

append(*extension*, *name=None*, **args*, ***kw*)
Adds a new top-level extension to the provider. Please, read the the concrete *DataProvider* documentation that is being used to know the exact behavior and additional accepted arguments.

Parameters

- **extension** (*array, Table, or other*) – The contents for the new extension. Usually the underlying *DataProvider* will understand how to deal with regular NumPy arrays and with AstroData *Table* instances, but it may also accept other types.
- **name** (*string, optional*) – A *DataProvider* will usually require a name for extensions. If the name cannot be derived from the metadata associated to *extension*, you will have to provider one.
- **args** (*optional*) – The DataProvider may accept additional arguments. Please, refer to its documentation.
- **kw** (*optional*) – The DataProvider may accept additional arguments. Please, refer to its documentation.

Returns

- *The instance that has been added internally (potentially *not* the same that*
- *was passed as *extension*)*

Raises

- TypeError – Will be raised if the *DataProvider* doesn't know how to deal with the data that has been passed.
- ValueError – Raised if the extension is of a proper type, but its value is illegal somehow.

descriptors
Returns a sequence of names for the methods that have been decorated as descriptors.

Returns

Return type A tuple of str

info()
Prints out information about the contents of this instance. Implemented by the derived classes.

load(*source*)
Class method that returns an instance of this same class, properly initialized with a DataProvider that can deal with the object passed as *source*

This method is abstract and has to be implemented by derived classes.

operate(*operator*, **args*, ***kwargs*)
Applies a function to the main data array on each extension, replacing the data with the result. The data will be passed as the first argument to the function.

It will be applied to the mask and variance of each extension, too, if they exist.

This is a convenience method, which is equivalent to:

for ext in ad: ad.ext.data = operator(ad.ext.data, *args, **kwargs) ad.ext.mask = operator(ad.ext.mask, *args, **kwargs) if ad.ext.mask is not None else None ad.ext.variance = operator(ad.ext.variance, *args, **kwargs) if ad.ext.variance is not None else None

with the additional advantage that it will work on single slices, too.

Parameters

- **operator** (*function, or bound method*) – A function that takes an array (and, maybe, other arguments) and returns an array
- **args** (*optional*) – Additional arguments to be passed positionally to the *operator*

- **kwargs** (*optional*) – Additional arguments to be passed by name to the *operator*

Examples

```
>>> import numpy as np
>>> ad.operate(np.squeeze)
```

reset(*data*, *mask=-23*, *variance=-23*, *check=True*)
Sets the .data, and optionally .mask and .variance attributes of a single-extension AstroData slice. This function will optionally check whether these attributes have the same shape.

Parameters

- **data** (*ndarray*) – The array to assign to the .data attribute ("SCI")
- **mask** (*ndarray, optional*) – The array to assign to the .mask attribute ("DQ")
- **variance** (*ndarray, optional*) – The array to assign to the .variance attribute ("VAR")
- **check** (*bool*) – If set, then the function will check that the mask and variance arrays have the same shape as the data array

Raises

- TypeError – if an attempt is made to set the .mask or .variance attributes with something other than an array
- ValueError – if the .mask or .variance attributes don't have the same shape as .data, OR if this is called on an AD instance that isn't a single extension slice

tags
A set of strings that represent the tags defining this instance

A.1.2 DataProvider

class astrodata.core.**DataProvider**
Abstract class describing the minimal interface that *DataProvider* derivative classes need to implement.

__getitem__(*slice*)
Returns a sliced view of the provider. It supports the standard Python indexing syntax, including negative indices.

Parameters **slice** (int, *slice*) – An integer or an instance of a Python standard *slice* object

Raises

- TypeError – If trying to slice an object when it doesn't make sense (eg. slicing a single slice)
- ValueError – If *slice* does not belong to one of the recognized types
- IndexError – If an index is out of range

Examples

```
>>> single = provider[0]
>>> multiple = provider[:5]
```

__iadd__(*oper*)
This method should attempt to do an in-place (modifying self) addition of each internal science object and the oper.

Parameters **oper** (*object*) – An operand to add to the internal science objects. The actual accepted type depends on the implementation

Returns

- Generally, it should return *self*. The implementations may decide to return
- *something else instead.*

__idiv__(*oper*)
This method should attempt to do an in-place (modifying self) division of each internal science object and the oper.

Parameters **oper** (*object*) – An operand to divide the internal science objects by. The actual accepted type depends on the implementation

Returns

- Generally, it should return *self*. The implementations may decide to return
- *something else instead.*

__imul__(*oper*)
This method should attempt to do an in-place (modifying self) multiplication of each internal science object and the oper.

Parameters **oper** (*object*) – An operand to multiply the internal science objects by. The actual accepted type depends on the implementation

Returns

- Generally, it should return *self*. The implementations may decide to return
- *something else instead.*

__isub__(*oper*)
This method should attempt to do an in-place (modifying self) subtraction of each internal science object and the oper.

Parameters **oper** (*object*) – An operand to subtract from the internal science objects. The actual accepted type depends on the implementation

Returns

- Generally, it should return *self*. The implementations may decide to return
- *something else instead.*

__len__()
"Length" of the object. This method will typically return the number of science objects contained by this provider, but this may change depending on the implementation.

Returns

Return type An integer

__weakref__
list of weak references to the object (if defined)

append(*ext*, *name=None*)
Adds a new component to the provider. Objects appended to a single slice will actually be made hierarchically dependent of the science object represented by that slice. If appended to the provider as a whole, the new member will be independent (eg. global table, new science object).

Parameters

- **ext** (array, *NDData*, *Table*, etc) – The component to be added. The exact accepted types depend on the class implementing this interface. Implementations specific to certain data formats may accept specialized types (eg. a FITS provider will accept an *ImageHDU* and extract the array out of it)
- **name** (*str, optional*) – A name that may be used to access the new object, as an attribute of the provider. The name is typically ignored for top-level (global) objects, and required for the others.

  It can consist in a combination of numbers and letters, with the restriction that the letters have to be all capital, and the first character cannot be a number ("[A-Z][A-Z0-9]*").

Returns

- *The same object, or a new one, if it was necessary to convert it to a more*
- *suitable format for internal use.*

Raises

- TypeError – If adding the object in an invalid situation (eg. *name* is *None* when adding to a single slice)
- ValueError – If adding an object that is not acceptable

data
A list of the the arrays (or single array, if this is a single slice) corresponding to the science data attached to each extension, in loading/appending order.

exposed
A collection of strings with the names of objects that can be accessed directly by name as attributes of this instance, and that are not part of its standard interface (ie. data objects that have been added dynamically).

Examples

```
>>> ad[0].exposed
set(['OBJMASK', 'OBJCAT'])
>>> ad[0].OBJCAT
...
```

is_settable(*attribute*)
Predicate that can be used to figure out if certain attribute of the *DataProvider* is meant to be modified by an external object.

This is used mostly by *AstroData*, which acts as a proxy exposing attributes of its assigned provider, to decide if it should set a value on the provider or on itself.

Parameters **attribute** (*str*) –

Returns

Return type A boolean

is_single
If this data provider represents a single slice out of a whole dataset, return True. Otherwise, return False.

Returns

Return type A boolean

is_sliced
If this data provider instance represents the whole dataset, return False. If it represents a slice out of the whole, return True.

Returns

Return type A boolean

mask
A list of the mask arrays (or a single array, if this is a single slice) attached to the science data, for each extension, in loading/appending order.

For objects that miss a mask, *None* will be provided instead.

uncertainty
A list of the uncertainty objects (or a single object, if this is a single slice) attached to the science data, for each extension, in loading/appending order.

The objects are instances of AstroPy's *NDUncertainty*, or *None* where no information is available.

See also:

variance The actual array supporting the uncertainty object

variance
A list of the variance arrays (or a single array, if this is a single slice) attached to the science data, for each extension, in loading/appending order.

For objects that miss uncertainty information, *None* will be provided instead.

See also:

uncertainty The *NDUncertainty* object used under the hood to propagate uncertainty when

operating

A.2 TagSet

class astrodata.core.**TagSet**(*add=None*, *remove=None*, *blocked_by=None*, *blocks=None*, *if_present=None*)
Named tuple that is used by tag methods to return which actions should be performed on a tag set. All the attributes are optional, and any combination of them can be used, allowing to create complex tag structures. Read the documentation on the tag-generating algorithm if you want to better understand the interactions.

The simplest TagSet, though, tends to just add tags to the global set.

It can be initialized by position, like any other tuple (the order of the arguments is the one in which the attributes are listed below). It can also be initialized by name.

add
Tags to be added to the global set

Type set of str, or None

remove
Tags to be removed from the global set

Type set of str, or None

blocked_by
Tags that will prevent this TagSet from being applied

Type set of str, or None

blocks
Other TagSets containing these won't be applied

Type set of str, or None

if_present
This TagSet will be applied only *all* of these tags are present

Type set of str, or None

Examples

```
>>> TagSet()
TagSet(add=set([]), remove=set([]), blocked_by=set([]), blocks=set([]), if_
↪present=set([]))
>>> TagSet(set(['BIAS', 'CAL']))
TagSet(add=set(['BIAS', 'CAL']), remove=set([]), blocked_by=set([]),␣
↪blocks=set([]), if_present=set([]))
>>> TagSet(remove=set(['BIAS', 'CAL']))
TagSet(add=set([]), remove=set(['BIAS', 'CAL']), blocked_by=set([]),␣
↪blocks=set([]), if_present=set([]))
```

A.3 NDAstroData

class astrodata.nddata.**NDAstroData**(*data*, *uncertainty=None*, *mask=None*, *wcs=None*, *meta=None*, *unit=None*, *copy=False*, *window=None*)

Implements NDData with all Mixins, plus some AstroData specifics.

This class implements an NDData-like container that supports reading and writing as implemented in the astropy.io.registry and also slicing (indexing) and simple arithmetics (add, subtract, divide and multiply).

A very important difference between NDAstroData and NDData is that the former attempts to load all its data lazily. There are also some important differences in the interface (eg. .data lets you reset its contents after initialization).

Documentation is provided where our class differs.

See also:

NDData, NDArithmeticMixin, NDSlicingMixin

Examples

The mixins allow operation that are not possible with NDData or NDDataBase, i.e. simple arithmetics:

```
>>> from astropy.nddata import NDAstroData, StdDevUncertainty
>>> import numpy as np
```

(continues on next page)

(continued from previous page)

```
>>> data = np.ones((3,3), dtype=np.float)
>>> ndd1 = NDAstroData(data, uncertainty=StdDevUncertainty(data))
>>> ndd2 = NDAstroData(data, uncertainty=StdDevUncertainty(data))

>>> ndd3 = ndd1.add(ndd2)
>>> ndd3.data
array([[ 2.,  2.,  2.],
       [ 2.,  2.,  2.],
       [ 2.,  2.,  2.]])
>>> ndd3.uncertainty.array
array([[ 1.41421356,  1.41421356,  1.41421356],
       [ 1.41421356,  1.41421356,  1.41421356],
       [ 1.41421356,  1.41421356,  1.41421356]])
```

see `NDArithmeticMixin` for a complete list of all supported arithmetic operations.

But also slicing (indexing) is possible:

```
>>> ndd4 = ndd3[1,:]
>>> ndd4.data
array([ 2.,  2.,  2.])
>>> ndd4.uncertainty.array
array([ 1.41421356,  1.41421356,  1.41421356])
```

See `NDSlicingMixin` for a description how slicing works (which attributes) are sliced.

data
: An array representing the raw data stored in this instance. It implements a setter.

set_section(*section*, *input*)
: Sets only a section of the data. This method is meant to prevent fragmentation in the Python heap, by reusing the internal structures instead of replacing them with new ones.

  Parameters

  - **section** (`slice`) – The area that will be replaced
  - **input** (`NDData`-like instance) – This object needs to implement at least `data`, `uncertainty`, and `mask`. Their entire contents will replace the data in the area defined by `section`.

Examples

```
>>> sec = NDData(np.zeros((100,100)))
>>> ad[0].nddata.set_section((slice(None,100),slice(None,100)), sec)
```

variance
: A convenience property to access the contents of `uncertainty`, squared (as the uncertainty data is stored as standard deviation).

window
: Interface to access a section of the data, using lazy access whenever possible.

  Returns

  - An instance of `NDWindowing`, which provides `__getitem__`, to allow the use
  - *of square brackets when specifying the window. Ultimately, an*

- `NDWindowingAstrodata` instance is returned

Examples

```
>>> ad[0].nddata.window[100:200, 100:200]
<NDWindowingAstrodata .....>
```

class `astrodata.nddata.`**NDWindowingAstroData**(*target*, *window*)

Allows "windowed" access to some properties of an `NDAstroData` instance. In particular, `data`, `uncertainty`, `variance`, and `mask` return clipped data.

A.4 Decorators and other helper functions

`astrodata.core.`**astro_data_descriptor**(*fn*)

Decorator that will mark a class method as an AstroData descriptor. Useful to produce list of descriptors, for example.

If used in combination with other decorators, this one *must* be the one on the top (ie. the last one applying). It doesn't modify the method in any other way.

Parameters **fn** (*method*) – The method to be decorated

Returns

Return type The tagged method (not a wrapper)

`astrodata.core.`**astro_data_tag**(*fn*)

Decorator that marks methods of an *AstroData* derived class as part of the tag-producing system.

It wraps the method around a function that will ensure a consistent return value: the wrapped method can return any sequence of sequences of strings, and they will be converted to a TagSet. If the wrapped method returns None, it will be turned into an empty TagSet.

Parameters **fn** (*method*) – The method to be decorated

Returns

Return type A wrapper function

`astrodata.core.`**returns_list**(*fn*)

Decorator to ensure that descriptors that should return a list (of one value per extension) only returns single values when operating on single slices; and vice versa.

This is a common case, and you can use the decorator to simplify the logic of your descriptors.

Parameters **fn** (*method*) – The method to be decorated

Returns

Return type A function

Index

Symbols

A

B

D

E

I

L

M