Astrodata Package Manual

Release 1.00

Craig Allen

March 09, 2012

CONTENTS

CHAPTER

ONE

INTRODUCTION

1.1 Document Brief

1.1.1 Revision History

- v1.0 - Document ready for internal review
- v0.91 - Document (nearly) ready for internal review
- v0.9 - Review document created
- v0.2 - Moved to Ophiuchus Craig Allen, Sphinx Generated
- v0.1 - (final) - Initial Revision, Craig Allen, Sphinx Generated, testing Sphinx

> < p>

1.1.2 Intended Audience

This document is intended for both new and experience developers using astrodata:

1. users of the astrodata package in conjunction with the "astrodata_Gemini" configuration package
2. developers creating new configuration information (types, descriptors, and transformations), e.g. instrument developers
3. potential developers needing to understand the work involved prior to development (e.g. for making proposals)
4. those trying to understand both what the system currently does, it's design philosophy, and where the package can or is expected to evolve

1.1.3 Document Stucture

This document is meant as an introductory user reference for Gemini Observatory's python-based data processing package, "astrodata". It is intended to serve both as an introductory reference for the actual function interfaces of two primary classes in the astrodata package, as well as a tool for new users to understand the general characteristics of the package. To this end this document contains three related but somewhat distinct sections:

- The first two chapters following this introduction are API reference manuals for the AstroData and ReductionContext classes respectively.
- A chapter on Creating an AstroData configuration Package, written as a hands on startup guide.
- A chapter on the Concepts in the AstroData Infrastructure

The AstroData class is a dataset abstraction for MEF files, while the ReductionContext is the interface for transformation primitives to communicate with the reduction system (i.e. access files in the pipeline, parameter information, execution context, and so on including all communication with the system.)

The astrodata package includes only the infrastructure code, but is generally shipped with the astrodata_Gemini configuration package, which contains all information and code regarding Gemini data types and type-specific transformations. The astrodata package also ships with an auxillary package of useful functions in the form of the "gempy" package.

The term "astrodata" in this document can refer to three somewhat distinct aspects of the system. There is "AstroData" the class, which is distinguishable in print by the camel caps capitalization and is the core software element of the system. There is "astrodata" the importable python package, which from the user's point of view imports the configurations which are available in the environment, but which strictly speaking contains only the infrustructural code. And there is simply "Astrodata" a loose term for the whole package, including the configuration package and support library.

CHAPTER

TWO

ASTRODATA CLASS REFERENCE

The following is information about the AstroData class. For descriptions of arguments shown for the class constructor, see AstroData.__init__(..). This documentation is generated from in source docstrings. To import the `AstroData` class use:

2.1 AstroData Class

from astrodata import AstroData

class `astrodata.data.`**`AstroData`**(*dataset=None*, *mode='readonly'*, *phu=None*, *header=None*, *data=None*, *store=None*, *storeClobber=False*, *exts=None*, *extInsts=None*)

The AstroData class abstracts datasets stored in MEF files and provides uniform interfaces for working on datasets from different instruments and modes. Configuration packages are used to describe the specific data characteristics, layout, and to store type-specific implementations.

MEFs can dbe generalized as lists of header-data units, with key-value pairs populating headers and pixel data populating data, AstroData interprets a MEF as a single complex entity. The individual "extensions" with the MEF are available using python list ("[]") syntax and are wrapped in AstroData objects (see `AstroData.__getitem__()`). AstroData uses pyfits for MEF I/O and numpy for pixel manipulations.

While the pyfits and numpy objects are available to the programmer, AstroData provides analagous methods for most pyfits functionality which allows it to maintain the dataset as a cohesive whole. The programmer does however use the numpy pixel arrays directly for pixel manipulation.

In order to identify types of dataset and provide type-specific behavior AstroData relies on configuration packages either in the PYTHONPATH environment variable or the Astrodata package environment variables, ADCONFIGPATH, or RECIPEPATH. The configuration (i.e. astrodata_Gemini) contains definitions for all instrument-mode-specific behavior. The configuration contains type definitions, meta-data functions, information lookup tables, and any other code or information needed to handle specific types of dataset.

This allows AstroData to manage access to the dataset for convienience and consistency. For example AstroData is able...:

- ... to allow reduction scripts to have easy access to dataset classification information in a consistent way across all instrument-modes
- ... to provide consistent interfaces for obtaining common meta-data across all instrument modes
- ... to relates internal extensions, e.g. discriminate between science and variance arrays and associate them properly
- ... to help propagate header-data units important to the given instrument mode, but unknown to general purpose transformations

AstroData's purpose in general is to provide smart dataset-oriented interfaces which adapt to dataset type. The primary interfaces of note are for file handling, dataset-type checking, and managing meta-data, but AstroData also integrates other functionality.

2.2 Basic Functions

2.2.1 AstroData Constructor

AstroData.**__init__**(*dataset=None*, *mode='readonly'*, *phu=None*, *header=None*, *data=None*, *store=None*, *storeClobber=False*, *exts=None*, *extInsts=None*)

Parameters

- **dataset** (*string, AstroData, HDUList*) – the dataset to load, either a filename (string) path or URL, an AstroData instance, or a pyfits.HDUList
- **mode** (*string*) – IO access mode, same as pyfits mode ("readonly", "update", "or append") with one additional AstroData-specific mode, "new". If the mode is "new", and a filename is provided, the constructor checks that the named file does not exist on disk, and if it does not it creates an empty AstroData of that name but does not write it to disk. Such an AstroData instance is ready to have HDUs appended, and to be written to disk at the user's command with "ad.write()".
- **phu** (*pyfits.core.Header*) – primary header unit. This object is propagated to all astrodata sub-data ImageHDUs. Special handling is made for header instances that are passed in as this arg., where a phu will be created and the '.header' will be assigned (ex. hdulist[0], ad.phu, ad[0].hdulist[0], ad['SCI',1].hdulist[0], ad[0].phu, ad['SCI',1].phu, and all the previous with .header appended)
- **header** – extension header for images (ex. hdulist[1].header, ad[0].hdulist[1].header, ad['SCI',1].hdulist[1].header)
- **data** (*numpy.ndarray*) – the image pixel array (ex. hdulist[1].data, ad[0].hdulist[1].data, ad['SCI',1].hdulist[1].data)
- **store** (*string*) – directory where a copy of the original file will be stored. This is used in the special case where the filename is an URL to a remote fits file. Otherwise it has no effect.
- **storeClobber** (*boolean*) – remote file handling for existing files with the same name. If true will save, if not, will delete.
- **exts** (*list*) – (advanced) a list of extension indexes in the parent HDUList that this instance should refer to, given integer or (EXTNAME, EXTVER) tuples specifying each extention in the "pyfits" index space where the PHU is at index 0, the first data extension is at index 1, and so on. I.e. This is primarilly intended for internal use creating "sub-data", which are AstroData instances that represent a slice, or subset, of some other AstroData instance.

  NOTE: if present this option will override and obscure the extInsts argument which will be ignored.

  Example of sub-data:

  sci_subdata = ad["SCI"]

  The sub-data is is created by passing "SCI" as an argument to the constructor. The 'sci_subdata' object would consist of its own AstroData instance referring to it's own HDUList, but the HDUs in this list would still be shared (in memory) with the 'ad' object, and appear in its HDUList as well.

- **extInsts** (*list of pyfits.HDU objects*) – (advanced) a list of extensions this instance should contain, given as actual pyfits.HDU instances. NOTE: if the "exts" argument is also set, this argument is ignored.

The AstroData constructor constructs an in-memory representation of a dataset. If given a filename it uses pyfits to open the dataset, reads the header and detects applicable types. Binary data, such as pixel data, is left on disk until referenced.

2.2.2 append(..)

`AstroData.`**`append`**(*moredata=None*, *data=None*, *header=None*, *auto_number=False*, *extname=None*, *extver=None*)

Parameters

- **moredata** (*pyfits.HDU, pyfits.HDUList, or AstroData*) – either an AstroData instance, an HDUList instance, or an HDU instance to add to this AstroData object. When present, data and header arguments will be ignored.
- **data** (*numarray.numaraycore.NumArray*) – if moredata *is not* specified, data and header should both be set and are used to construct a new HDU which is then added to the AstroData object. The 'data' argument should be set to a valid numpy array.
- **header** (*pyfits.Header*) – if moredata *is not* specified, data and header are used to make an HDU which is then added to the HDUList associated with this AstroData instance. The 'header' argument should be set to a valid pyfits.Header object.
- **auto_number** (*boolean*) – auto-increment extver to fit file convention
- **extname** (*int*) – extension name (ex, 'SCI', 'VAR', 'DQ')
- **extver** – extension's "extver" value

This function appends more header-data units (aka "HDUs") to the AstroData instance.

2.2.3 close(..)

`AstroData.`**`close`**()

The close(..) function will close the HDUList associated with this AstroData instance.

2.2.4 insert(..)

`AstroData.`**`insert`**(*index*, *moredata=None*, *data=None*, *header=None*, *auto_number=False*, *extname=None*, *extver=False*)

Parameters

- **index** (*integer or (EXTNAME,EXTVER) tuple*) – the extension index, either an int or (EXTNAME, EXTVER) pair before which the extension is to be inserted. Note, the first data extension is [0], you cannot insert before the PHU. Index always refers to Astrodata Numbering system, 0 = HDU
- **moredata** (*pyfits.HDU, pyfits.HDUList, or AstroData*) – Either an AstroData instance, an HDUList instance, or an HDU instance. When present, data and header will be ignored.
- **data** (*numarray.numaraycore.NumArray*) – if moredata *is not* specified, data and header should both be set and are used to construct a new HDU which is then added to the AstroData instance.

- **header** (*pyfits.Header*) – if moredata *is not* specified, data and header are used to make an HDU which is then added to the HDUList associated with this AstroData instance.
- **auto_number** (*boolean*) – auto-increment appends to match existing extname - extver convention.
- **extname** (*string*) – extension name (ex, 'SCI', 'VAR', 'DQ')
- **extver** (*integer*) – extension version (ex, 1, 2, 3)

This function inserts more data units (aka an "HDU") to the AstroData instance.

2.2.5 info(..)

`AstroData.`**`info`**(*oid=False*, *table=False*, *help=False*)

The info(..) function prints self.infostr() and is maintained for convienience and low level debugging.

2.2.6 infostr(..)

`AstroData.`**`infostr`**(*as_html=False*, *oid=False*, *table=False*, *help=False*)

Parameters

- **as_html** (*bool*) – return as HTML formatted string
- **oid** (*bool*) – include object id
- **help** (*bool*) – include sub-data reference information

The infostr(..) function is used to get a string ready for display either as plain text or HTML. It provides AstroData-relative information, unlike the pyfits-forwarded function AstroData.info(), and so uses AstroData relative indexes, descriptors, and so on.

2.2.7 write(..)

`AstroData.`**`write`**(*filename=None*, *clobber=False*, *rename=None*)

Parameters

- **fname** (*string*) – file name to write to, optional if instance already has name, which might not be the case for new AstroData instances created in memory.
- **clobber** (*bool*) – This flag drives if AstroData will overwrite an existing file.
- **rename** (*bool*) – This flag allows you to write the AstroData instance to a new filename, but leave the "current" name in tact in memory.

The write function acts similarly to the pyfits HDUList.writeto(..) function if a filename is given, or like pyfits.HDUList.update(..) if no name is given, using whatever the current name is set to. When a name is given, this becomes the new name of the AstroData object and will be used on subsequent calls to write for which a filename is not provided. If the clobber flag is False (the default) then write(..) throws an exception if the file already exists.

2.3 Type Information

`AstroData.`**`is_type`**(**typenames*)

Parameters **typename** (*string*) – specifies the type name to check.

Returns True if the given types all apply to this dataset, False otherwise

Return type Bool

This function checks the AstroData object to see if it is the given type(s) and returns True if so.

Note "AstroData.check_type(..)" is an alias for "AstroData.is_type(..)".

`AstroData.`**`get_types`**(*prune=False*)

Parameters **prune** (*bool*) – flag which controls 'pruning' the returned type list so that only the leaf node type for a given set of related types is returned.

Returns a list of classification names that apply to this data

Return type list of strings

The get_types(..) function returns a list of type names, where type names are as always, strings. It is possible to "prune" the list so that only leaf nodes are returned, which is useful when leaf nodes take precedence such as descriptors.

Note: types are divided into two categories, one intended for types which represent processing status (i.e. RAW vs PREPARED), and another which contains a more traditional "typology" consisting of a heirarchical tree of dataset types. This latter tree maps roughly to instrument-modes, with instrument types branching from the general observatory type, (e.g. "GEMINI").

To retrieve only status types, use get_status(..); to retreive just typological types use get_typology(..). Note that the system does not enforce what checks are actually performed by types in each category, that is, one could miscategorize a type when authoring a configuration package. Both classifications use the same DataClassification objects to classify datasets. It is up to those implementing the type-specific configuration package to ensure types related to status appear in the correct part of the configuration space.

Currently the distinction betwen status and typology is not used by the system (e.g. in type-specific default recipe assignments) and is provided as a service for higher level code, e.g. primitives and scripts which make use of the distinction.

`AstroData.`**`get_status`**(*prune=False*)

This function returns the set of type names (strings) which apply to this dataset and which come from the status section of the AstroData Type library. "Status" classifications are those which tend to change during the reduction of a dataset based on the amount of processing, e.g. RAW vs PREPARED. Strictly, a "status" type is any type defined in or below the status part of the `classification` directory within the configuration. I.e. in the Gemini type configuration any type definition files in or below the "astrodata_Gemini/ADCONFIG/classification/status" directory.

Returns a list of string classification names

Return type list of strings

`AstroData.`**`get_typology`**`()`

This function returns the set of type names (strings) which apply to this dataset and which come from the typology section of the AstroData Type library. "Typology" classifications are those which tend to remain with the data in spite of reduction status, e.g. those related to the instrument-mode of the dataset or of the datasets used to produce it. Strictly these consist of any type defined in or below the correct configuration directory, i.e. in Gemini's configuration, "astrodata_Gemini/ADCONFIG/classification/types" directory.

Returns a list of classification name strings

Return type list of strings

2.4 Header Manipulations

Manipulations of headers, specifically retrieving and setting key-value pair settings in the header section of header-data units can be done directly using the AstroData header manipulation functions which cover both PHU and extension headers. For higher level metadata which is available for all types in the tree in a properly constructed configuration space, the metadata is retreived by descriptor functions, accessed as members of the AstroData object.

To set information or retrieve meta-data not covered by descriptors, one must read and write key-value pairs to the HDU headers at the lower-level. AstroData offers three pairs of functions for getting and setting header values, for each of three distinct cases. While it is possible to use the pyfits.Header directly (available via "ad[..].header"), it is preferrable to use the AstroData calls which allow AstroData to keep type information up to date, as well as to update any other characteristics of the AstroData object which may need to be maintained when the dataset is changed.

The three distinct pairs of header access functions serve the following purposes:

- set/get headers in PHU.
- set/get headers in the single extension of a "single-HDU AstroData object".
- set/get headers in an extension of a multi-HDU (aka "multi-extension") AstroData instance. This requires specifying the extension index, and cannot be used to modify the PHU. HDU #0 is the first real header-data section in the MEF.

2.4.1 Set/Get PHU Headers

`AstroData.`**`phu_get_key_value`**(*key*)

Parameters **key** (*string*) – name of header value to retrieve

Return type string

Returns the key's value as string or None if not present.

The phu_get_key_value(..) function returns the value associated with the given key within the primary header unit of the dataset. The value is returned as a string (storage format) and must be converted as necessary by the caller.

`AstroData.`**`phu_set_key_value`**(*key*, *value*, *comment=None*)

Parameters

- **key** (*string*) – name of PHU header value to set
- **value** (*string (or can be converted to string)*) – value to apply to PHU header
- **comment** (*string*) – value to be put in the comment part of the header key

The phu_set_key_value(..) function is used to set the value (and optionally the comment) associated with a given key in the primary header unit of the dataset. The value argument will be converted to string, so it must have a string operator member function or be passed in as string.

2.4.2 Set/Get Single-HDU Headers

`AstroData.`**`get_key_value`**(*key*)

Parameters **key** (*string*) – name of header value to set

Returns the specified value

Return type string

The get_key_value(..) function is used to get the value associated with a given key in the data-header unit of a single-HDU AstroData instance (such as returned by iteration). The value argument will be converted to string, so it must have a string operator member function or be passed in as string.

Note Single extension AstroData objects are those with only a single header-data unit besides the PHU. They may exist if a single extension file is loaded, but in general are produced by indexing or iteration instructions, i.e.:

sead = ad[("SCI",1)]

for sead in ad["SCI"]: ...

The variable "sead" above is ensured to hold a single extension AstroData object, and can be used more convieniently.

AstroData.**set_key_value**(*key*, *value*, *comment=None*)

Parameters

- **key** (*string*) – name of data header value to set
- **value** (*string (or can be converted to string)*) – value to apply to header
- **comment** (*string*) – value to be put in the comment part of the header key

The set_key_value(..) function is used to set the value (and optionally the comment) associated with a given key in the data-header of a single-HDU AstroData instance.

Note Single extension AstroData objects are those with only a single header-data unit besides the PHU. They may exist if a single extension file is loaded, but in general are produced by indexing or iteration instructions, i.e.:

sead = ad[("SCI",1)]

for sead in ad["SCI"]: ...

The variable "sead" above is ensured to hold a single extension AstroData object, and can be used more convieniently.

2.4.3 Set/Get Multiple-HDU Headers

AstroData.**ext_get_key_value**(*extension*, *key*)

Parameters

- **extension** (*int or (EXTNAME, EXTVER) tuple*) – identifies which extension, either an integer index or (EXTNAME, EXTVER) tuple
- **key** (*string*) – name of header entry to retrieve

Return type string

Returns the value associated with the key, or None if not present

This function returns the value from the given extension's header, with "0" being the first data extension. To get values from the PHU use phu_get_key_value(..).

AstroData.**ext_set_key_value**(*extension*, *key*, *value*, *comment=None*)

Parameters

- **extension** (*int or (EXTNAME, EXTVER) tuple*) – identifies which extension, either an integer index or (EXTNAME, EXTVER) tuple
- **key** (*string*) – name of PHU header value to set

- **value** (*string (or can be converted to string)*) – value to apply to PHU header
- **comment** (*string*) – value to be put in the comment part of the header key

The ext_set_key_value(..) function is used to set the value (and optionally the comment) associated with a given key in the header unit of the given extension within the dataset. This function sets the value in the given extension's header, with "0" being the first data extension. To set values in the PHU use phusetKeyValue(..).

2.5 Iteration and Subdata

2.5.1 Overview

Using Slices and "Subdata"

<html><

AstroData instances are presented as lists of AstroData instances. However, internally the list is merely a list of extensions and the *AstroData.getitem(..)* function (which implements the "[]" syntax) creates AstroData instances on the fly when called. Such instances share information in memory with their parent instance. This is the in line with the general operation of pyfits and numpy, and in general how Python handles objects. This allows efficient use of memory and disk I/O. To make copies one must explicitly ask for copies. Thus when one takes a slice of a numpy array, that slice, although possibly of a different dimensionality and certainly of range, is really just a view onto the original memory, changes to the slice affect the original. If one takes a subset of an AstroData instance's HDUList, then the save HDUs are present in both the original and the sub-data. To make a separate copy one must use the *deepcopy* built-in function (see below).

As the diagram indicates, when taking a subset of data from an AstroData instance using the square brackets operator, you receive a newly created AstroData instance which is associated only with those HDUs identified. Changes to a shared HDU's data or header member will be reflected in both AstroData instances. Generally speaking this is what you want for efficient operation. If you do want to have entirely separate data, such that changes to the data sections of one do not affect the other, use the python deepcopy operator:

```
from copy import deepcopy

ad = AstroData("dataset.fits")
scicopy = deepcopy(ad["SCI"])
```

If on the other hand all you want is to avoid changing the original dataset on disk, and do not need the original data, untransformed, in memory along with the transformed version, which is the usual case, then you can write the AstroData subdata instance to a new filename:

```
from astrodata import AstroData

ad = AstroData("dataset.fits")
scicopy = ad["SCI"]
scicopy.write("datasetSCI.fits")
```

2.5.2 countExts(..)

AstroData.**count_exts**(*extname*)

Parameters **extname** (*string*) – the name of the extension, equivalent to the value associated with the "EXTNAME" key in the extension header.

Returns number of extensions of that name

Return type int

The count_exts(..) function counts the extensions of a given name (as stored in the HDUs "EXTVER" header).

2.5.3 The [] Operator

`AstroData.__getitem__(ext)`

Parameters **ext** (*string, int, or tuple*) – The integeter index, an indexing `(EXTNAME, EXTVER)` tuple, or EXTNAME name. If an int or tuple, the single extension identified is wrapped with an AstroData instance, and "single-extension" members of the AstroData object can be used. If a string, `EXTNAME`, is given, then all extensions with the given EXTNAME will be wrapped by the new AstroData instance.

Returns an AstroData instance associated with the subset of data.

Return type AstroData

This function supports the "[]" syntax for AstroData instances, e.g. *ad[("SCI",1)]*. We use it to create AstroData objects associated with "subdata" of the parent AstroData object, that is, consisting of an HDUList which consists of some subset of the parent MEF. e.g.:

```
datasetA = AstroData.AstroData("datasetMEF.fits")
datasetB = datasetA[SCI]
```

In this case, after the operations, datasetB is an AstroData object associated with the same MEF, sharing some of the the same actual HDUs in memory as datasetA. The object in "datasetB" will behave as if the SCI extensions are its only members, and it does in fact have its own pyfits.HDUList. Note that datasetA and datasetB share the PHU and also the data structures of the HDUs they have in common, so that a change to "datasetA[('SCI',1)].data" will change the "datasetB[('SCI',1)].data" member and vice versa. They are in fact both references to the same numpy array in memory. The HDUList is a different list, however, that references common HDUs. If a subdata related AstroData object is written to disk, the resulting MEF will contain only the extensions in the subdata's HDUList.

Note Integer extensions start at 0 for the data-containing extensions, not at the PHU as with pyfits. This is important: ad[0] is the first content extension, in a traditional MEF perspective, the extension AFTER the PHU; it is not the PHU! In AstroData instances, the PHU is purely a header, and not counted as an extension in the way that headers generally are not counted as their own elements in the array they contain meta-data for. The PHU can be accessed via the `phu` AstroData member of using the PHU related member functions.

2.6 Single HDU AstroData Attributes

2.6.1 data attribute

`AstroData.data`

The data property can only be used for single-HDU AstroData instances, such as those returned during iteration. It is a property attribute which uses *get_data(..)* and *set_data(..)* to access the data members with "=" syntax. To set the data member, use *ad.data = newdata*, where *newdata* must be a numpy array. To get the data member, use *npdata = ad.data*.

`AstroData.get_data()`

Returns data array associated with the single extension

Return type pyfits.ndarray

The *get_data(..)* member is the function behind the property-style "data" member and returns appropriate HDU's data member(s) specifically for the case in which the AstroData instance has ONE HDU (in addition to the PHU). This allows a single-extension AstroData, such as AstroData generates through iteration, to be used as though it simply is just the one extension, e.g. allowing *gd.data* to be used in place of the more esoteric and ultimately more dangerous *gd[0].data*. One can assure one is dealing with single extension AstroData instances when iterating over the AstroData extensions and when picking out an extension by integer or tuple indexing, e.g.:

```
for gd in dataset[SCI]:
    # gd is a single-HDU index
    gd.data = newdata

# assuming the named extension exists,
# sd will be a single-HDU AstroData
sd = dataset[("SCI",1)]
```

AstroData.**set_data**(*newdata*)

Parameters **newdata** (*numarray.numarraycore.NumArray*) – new data objects

Raises **Errors.SingleHDUMemberExcept** if AstroData instance has more than one extension (not including PHU).

This function sets the data member of a data section of an AstroData object, specifically for the case in which the AstroData instance has ONE header-data unit (in addition to PHU). This case is assured when iterating over the AstroData extensions, e.g.:

```
for gd in dataset[SCI]:
    ...
```

2.6.2 header attribute

AstroData.**header**

The header property can only be used for single-HDU AstroData instances, such as those returned during iteration. It is a property attribute which uses *get_header(..)* and *set_header(..)* to access the header member with the "=" syntax. To set the header member, use *ad.header = newheader*, where *newheader* must be a pyfits.Header object. To get the header member, use *hduheader = ad.header*.

AstroData.**get_header**(*extension=None*)

Returns header

Return type pyfits.Header

Raises **Errors.SingleHDUMemberExcept** Will raise an exception if more than one extension exists. (note: The PHU is not considered an extension in this case)

The get_header(..) function returns the header member for Single-HDU AstroData instances (which are those that have only one extension plus PHU). This case can be assured when iterating over extensions using AstroData, e.g.:

```
for gd in dataset[SCI]:
    ...
```

AstroData.**set_header**(*header*, *extension=None*)

Parameters

- **header** (*pyfits.Header*) – pyfits Header to set for given extension
- **extension** (*int or tuple, pyfits compatible extension index*) – Extension index to retrieve header, if None or not present then this must be a single extension AstroData instance,

which contains just the PHU and a single data extension, and the data extension's header is returned.

Raises Errors.SingleHDUMemberExcept Will raise an exception if more than one extension exists.

The set_header(..) function sets the extension header member for single extension (which are those that have only one extension plus PHU). This case is assured when iterating over extensions using AstroData, e.g.:

for gd in dataset[SCI]: ...

2.6.3 Renaming an Extension

AstroData.**rename_ext**(*name*, *ver=None*, *force=True*)

Parameters

- **name** (*string*) – New "EXTNAME" for the given extension.
- **ver** (*int*) – New "EXTVER" for the given extension

Note: This member only works on single extension AstroData instances.

The rename_ext() function is used in order to rename an HDU with a new EXTNAME and EXTVER based identifier. Merely changing the EXTNAME and EXTEVER values in the extensions pyfits.Header are not sufficient. Though the values change in the pyfits.Header object, there are special HDU class members which are not updated.

Warning This function maniplates private (or somewhat private) HDU members, specifically "name" and "_extver". STSCI has been informed of the issue and has made a special HDU function for performing the renaming. When generally available, this new function will be used instead of manipulating the HDU's properties directly, and this function will call the new pyfits.HDUList(..) function.

2.7 Module Level Functions

2.7.1 correlate(..)

astrodata.data.**correlate**(**iary*)

Parameters **iary** (*list of AstroData instance*) – A list of AstroData instances for which a correlation dictionary will be constructed to produce a correlation dict.

Returns a list of tuples containing correlated extensions from the arguments.

Return type list of tuples

The *correlate(..)* function is a module-level helper function which returns a list of tuples of Single Extension AstroData instances which associate extensions from each listed AstroData object, to identically named extensions among the rest of the input array. The *correlate(..)* function accepts a variable number of arguments, all of which should be AstroData instances.

The function returns a structured dictionary of dictories of lists of AstroData objects. For example, given three inputs, *ad*, *bd* and *cd*, all with three "SCI", "VAR" and "DQ" extensions. Given *adlist = [ad, bd, cd]*, then *corstruct = correlate(adlist)* will return to *corstruct* a dictionary first keyed by the EXTNAME, then keyed by tuple. The contents (e.g. of *corstruct["SCI"][1]*) are just a list of AstroData instances each containing a header-data unit from *ad*, *bd*, and *cd* respectively.

Info to appear in the list, all the given arguments must have an extension with the given (EXTNAME,EXTVER) for that tuple.

2.7.2 prep_output(..)

`astrodata.data.`**`prep_output`**(*input_ary=None*, *name=None*, *clobber=False*)

Parameters

- **input_ary** (*list of AstroData Instances*) – The input array from which propagated content (such as the source PHU) will be taken. Note: the zero-th element in the list is used as the reference dataset, for PHU or other items which require a particular reference.
- **name** – File name to use for returned AstroData, optional.
- **clobber** (*bool*) – By default prep_output(..) checks to see if a file of the given name already exists, and will raise an exception if found. Set *clobber* to *True* to override this behavior and potentially overwrite the extant file. The datset on disk will not be overwritten as a direct result of prep_output, which only prepares the object in memory, but will occur when the AstroData object returned is written (i.e. *ad.write()*).

Returns an AstroData instance initialized with appropriate header-data units such as the PHU, Standard Gemini headers and with type-specific associated data-header units such as binary table Mask Definition tables (aka MDF).

Return type AstroData

..info: File will not have been written to disk by prep_output(..).

The prep_output(..) function creates a new AstroData object ready for appending output information (e.g. `ad.append(..)`). While you can also create an empty AstroData object by giving no arguments to the AstroData constructor (i.e. `ad = AstroData()`), `prep_output(..)` exists for the common case where a new dataset object is intended as the output of come combinatorial process on a list of source dataset, and some information from the source inputs must be propagated.

The `prep_output(..)` function makes use of this knowledge to ensure the file meets standards in what is considered a complete output file given such a combination. In the future this function can make use of dataset history and structure definitions in the ADCONFIG configuration space. As `prepOutput` improves, scripts and primitives that use it will benefit in a forward compatible way, in that their output datasets will benefit from more automatic propagation, validations, and data flow control, such as the emergence of history database propagation.

Presently, it already provides the following:

•Ensures that all standard headers are in place in the new file, using the configuration .

•Copy the PHU of the reference image (input_ary[0]).

•**Propagate associated information such as the MDF in the case of a MOS** observation, configurable by the Astrodata Structures system.

2.7.3 re_header_keys(..)

`astrodata.data.`**`re_header_keys`**(*rekey*, *header*)

Parameters

- **rekey** (*string*) – a regular expresion to match to keys in header
- **header** (*pyfits.Header*) – a pyfits.Header object as returned by *ad[("SCI",1)].header*

Returns a list of keys that appear in the given header

Return type list of strings

This utility function returns a list of keys from the header passed in which match the given regular expression.

CHAPTER

THREE

REDUCTIONCONTEXT CLASS REFERENCE

The following is information about the ReductionContext class. When writing primitives the reduction context is passed into the primitive as the the sole argument (generally named `rc` by Gemini conventions and in addition to the `self` argument). This object is used by the primitive to both get inputs and store outputs, as well as to communicate with subsystems like the calibration queries system or list keeping for stacking.

3.1 Parameter and Dictionary Features

3.1.1 The "in" operator: contains(..)

`ReductionContext.__contains__`(*thing*)

Parameters **thing** (*str*) – A key to check for presences in the Reduction Context

The __contains__ function implements the python 'in' operator. The ReductionContext is a subclass of a `dict`, but it also has a secondary dict of "local parameters" which are available to the current primitive only, which are also tested by the `__contains__(..)` member. These parameters will generally be those passed in as arguments to a primitive call from a recipe.

3.2 Dataset Streams: Input and Output Datasets

3.2.1 get_inputs(..)

`ReductionContext.get_inputs`(*style=None*)

Parameters **style** (*string*) – Controls the type of return value. Supported values are "AD" and "FN" for `AstroData` and `string` filenames respectively.

Returns a list of `AstroData` instances or `string` filenames

Return type list

Get inputs gets the current input datasets from the current stream. You cannot choose the stream, use `get_stream(..)` for that. To report modified datasets back to the stream use `report_output(..)`.

3.2.2 get_inputs_as_astrodata(..)

ReductionContext.**get_inputs_as_astrodata**()

This function is equivalent to:

```
get_inputs(style="AD")
```

3.2.3 get_inputs_as_filenames(..)

ReductionContext.**get_inputs_as_filenames**()

This function is equivalent for:

```
get_inputs(style="FN")
```

3.2.4 get_stream(..)

ReductionContext.**get_stream**(*stream='main'*, *empty=False*, *style=None*)

Parameters

- **stream** (*str*) – A string name for the stream in question. To use the standard stream do not set.
- **empty** (*bool*) – Controls if the stream is emptied, defaults to "False".
- **style** – controls the type of output. "AD" directs the function to return a list of AstroData instances. "FN" directs it to return a list of filenames. If left blank or set to `None`, the AstroDataRecord structures used by the Reduction Context will be returned.

Returns a list of datasets as `AstroData` or filenames.

Return type list

Get stream returns a list of AstroData instances in the given stream.

3.2.5 get_reference_image(..)

ReductionContext.**get_reference_image**()

This function returns the current reference image. At the moment this is simply the first dataset in the current inputs. However, use of this function allows us to evolve our concept of reference image for more complicated issues where choice of a "reference" image may be more complicated (e.g. require some data analysis to determine).

3.2.6 report_output(..)

ReductionContext.**report_output**(*inp*, *stream=None*, *load=True*)

Parameters

- **inp** (*str, AstroData instance, or list*) – The inputs to report (add to the given or current stream). Input can be a string (filename), an AstroData instance, or a list of strings and/or AstroData instances. Each individual dataset is wrapped in an AstroDataRecord and stored in the current stream.
- **stream** (*str*) – If not specified the default ("main") stream is used. When specified the named stream is created if necessary.

- **load** – A boolean (default: True) which specifies whether string arguments (pathnames) should be loaded into AstroData instances or if it should be kept as a filename, unloaded. This argument has no effect when "report" `AstroData` instances already in memory.

This function, along with `get_inputs(..)` allows a primitive to interact with the datastream in which it was invoked (or access other streams).

3.2.7 switch_stream(..)

`ReductionContext.`**`switch_stream`**(*switch_to=None*)

Parameters **switch_to** (*str*) – The string name of the stream to switch to. The named stream must already exist.

Note This function is used by the infrastructure (in an application such as reduce and in the ReductionContext) to switch the stream being used. Reported output then goes to the specified stream.

3.3 ADCC Services

3.4 Calibrations

3.4.1 get_cal(..)

`ReductionContext.`**`get_cal`**(*data*, *caltype*)

Retrieve calibration.

Parameters

- **data** (*str or AstroData instance*) – File for which calibration will be applied.
- **caltype** (*str*) – The type of calibration (ex.'bias', 'flat').

Returns The URI of the currently stored calibration or None.

Return type str or None

3.4.2 rq_cal(..)

`ReductionContext.`**`rq_cal`**(*caltype*, *inputs=None*, *source='all'*)

Create calibration requests based on raw inputs.

Parameters

- **caltype** (*str*) – The type of calibration. For example, 'bias' and 'flat'.
- **inputs** (*list of AstroData instances*) – The datasets for which to find calibrations, if not present or `None` current "inputs" are used.
- **source** – Directs what calibration service to contact, for future compatibility, surrently only "all" is supported.

3.5 Stacking

3.5.1 rq_stack_get(..)

ReductionContext.**rq_stack_get**(*purpose=''*)

Parameters **purpose** (*str*) – The purpose is a string prepended to the stackingID used to identify the list (see get_list).

The stackingID (see IDFactory module) is used to identify the list. The first input in the rc.inputs list is used as the reference image to generate the stackingID portion of the list identifier.

The stackingID function in IDFactory is meant to produce identical stacking identifiers for different images which can/should be stacked together, e.g. based on program id and/or other details. Again, see IDFactory for the particular algorithm in use.

Note a versioning system is latent within the code, and is added to the id to allow adaptation in the future if identifer construction methods change.

3.5.2 rq_stack_update(..)

ReductionContext.**rq_stack_update**(*purpose=None*)

Parameters **purpose** (*str*) – The purpose argument is a string prefixed to the generated stackingID. This allows two images which would produce identical stackingIDs to go in different lists, i.e. such as a fringe frame which, which might be prepended with "fringe" as the purpose.

This function creates requests to update a stack list with the files in the current rc.inputs list. Each will go in a stack based on its own stackingID (prepended with "purpose").

Note this function places a message on an outbound message queue which will not be sent until the next "yield", allowing the ReductionObject command clause to execute.

3.6 Lists

3.6.1 list_append(..)

ReductionContext.**list_append**(*id*, *files*, *cachefile=None*)

Parameters

- **id** (*str*) – A string which identifies the list to append the listed filenames to.
- **files** (*list of str*) – A list of filenames to add to the list.
- **cachefile** (*str*) – The filename to use to store the list.

The caller is expected to supply cachefile, though in principle a value of "None" could mean the "default cachefile" this is not supported by the adcc as of yet, since the desired behavior is for reduce instances running in the same directory to cooperate, and those running in separate directories be kept separate, and this is implemented by providing an argument for cachefile which is in a generated subdirectory (hidden) based on the startup directory for the reduce process.

The adcc will negotiate all contention and race conditions regarding multiple applications manipulating a list simultaneously in separate process.

3.6.2 get_list(..)

ReductionContext.**get_list**(*id*)

Parameters **id** (*str*) – Lists are assiciated with arbitrary identifiers, passed as strings. See IDFactory for ids built from standard astrodata characteristics.

The list functionality allows storing dataset names in a list which is shared by all instances of reduce running in a given directory. The list is kept by an adcc instance in charge of that sub-directory. The "get_list" function retrieves a list that has already been requested via "rq_stack_get()" which initiates the interprocess request.

This function does not block, and if the stack was not requested prior to a yeild, prior to this call, then None or an out of date version of this list will be retrieved.

Note "get_stack" calls get_list but takes a "purpose" to which it adds a stackingID as a suffix to the list identifier.

3.7 Utility

3.7.1 prepend_names(..)

ReductionContext.**prepend_names**(*prepend*, *current_dir=True*, *filepaths=None*)

Parameters

- **prepend** (*string*) – The string to be put at the front of the file.
- **current_dir** (*boolean*) – Used if the filename (astrodata filename) is in the current working directory.
- **filepaths** – If present, these file paths will be modified, otherwise the current inputs are modified.

Returns List of new prepended paths.

Return type list

Prepends a prefix string to either the inputs or the given list of filenamesfilename.

3.7.2 run(..)

ReductionContext.**run**(*stepname*)

Parameters **stepname** (*string*) – The primitive or recipe name to run. Note: this is actually compiled as a recipe... proxy recipe names may appear in the logs.

The `run(..)` function allows a primitive to use the reduction context to execute another recipe or primitive.

CHAPTER

FOUR

ASTRODATA CONFIGURATION PACKAGE DEVELOPMENT GUIDE

4.1 Elements

Instrument-mode-specific behaviors available through the AstroData class are not implemented in the astrodata package itself, but are instead loaded from configuration packages. In the case of Gemini data the configuration package is a directory named `astrodata_Gemini`. This configuration path is found by astrodata by the containing directory appearing either on the PYTHONPATH, or on either of two astrodata environment variables, RECIPEPATH or ADCONFIGPATH.

The astrodata package searches for all directories named `astrodata_<anything>` in these environment variables. Though the configurations contain exectable python, it is not meant to be imported as a regular python module but is loaded by the astrodata package.

4.1.1 The General Configuration Creation Process

1. You will define a tree of AstroDataTypes identifying types of your data.
2. You will create "descriptor" functions which calculate a particular metadata value for nodes of the AstroDataType tree defined, such as `gain` or `filter_name`.
3. You will write python member functions bundled into PrimitivesSet classes, which specifically understand your dataset.
4. You will assemble primitives into sequential lists, which we call processing "recipes".

Initially you will develop classifications for your data, and functions which will provide standard information, allowing you to use AstroData, e.g. in processing scripts. Then you will put your processing scripts in the form of "primitives" and collect these in "recipes" so they can be used for automated data reduction.

4.1.2 Configuration Elements Which Have To Be Developed

1. **AstroData Types** identify classifications of MEF dataset to which other features can be assigned. Types have requirements which must hold for an identified dataset and also information about the place of the type in an overall type hierarchy (e.g. The GMOS type is the parent of GMOS_IMAGE).
2. **AstroData Descriptors** are functions which calculate a particular type of metadata which is expected to be available for all datasets throughout the type hierarchy. Examples from the Gemini configuration package are `gain` and `filtername`. Different instruments store information about the gain in unique headers, and may

even require lookup tables not located in the dataset. Descriptors are type-appropriate functions assigned at runtime to the astrodata instance, allowing type-specific implementations to manage these peculiarities.

3. **Primitives** are dataset transformations meant to run in the recipe system. Primitives are implemented as python generator functions in sets of primitives that apply to a common AstroDataType.
4. **Recipes** are lists of primitives stored in plain text which can be executed by the AstroData Recipe System. While primitives work on the `Reduction Context` explicitely, the reduction context is implicit in recipes so that recipes can arguably be considered to contain "scientifically meaningful" steps with no "software artifacts".

4.2 Creating A Config Package

4.2.1 Preparation

For this work it is required that the Gemini Python AstroData package already be installed and functional. In general the Gemini AstroData configuration package(s) will already be installed somewhere on the PYTHONPATH, and Gemini Python scripts such as `reduce` and `typewalk` will be on the system path. Installations from SVN will require that the `astrodata/scripts` package directory be added to the PATH.

4.2.2 Clone the Sample Package

The easiest and recommended way to start a new configuration package is by copying the `astrodata_Sample` package. The sample configuration package is located in `astrodata/sample/astrodata_Sample`. Copy this directory to a development workspace as in the following example, where `<ad_install_dir>` should be the directory in which the `astrodata` package is installed:

```
cd /home/username
mkdir workspace
cd workspace

cp -r <ad_install_dir>/astrodata/samples/astrodata_Sample .
```

Note that presumably, `<ad_install_dir>` should already be on the PYTHONPATH.

The name of the destination can, of course, be other than `astrodata_Sample`, and it can be changed later as well. For a real package it must be changed, and though not strictly necessary, the `ADCONFIG_Sample` and `RECIPES_Sample` should have "Sample" changed to something unique which matches the parent `astrodata_<whatever>` directory. So long as the `ADCONFIG_` and `RECIPIES_` portion of the name is present no other aspect of the configuration will have to change. However, every configuration package wherever on the path must have a unique name.

You must also ensure the the new directory *containing* `astrodata_Sample` is in either ADCONFIGPATH or RECIPEPATH, or alternately for convienience (i.e. when installing packages via setup.py) in the PYTHONPATH. If you are following the above steps, you are in the directory to which `astrodata_Sample` was copied. Add this directory to the RECIPEPATH so your copy of `astrodata_Sample` can be found:

```
export RECIPEPATH=$(pwd):$RECIPEPATH
```

You can now test that `astrodata_Sample` is being discovered by the astrodata package by running a tool from the `astrodata/scripts` directory which should have been installed to system bin directories by the `setup.py` process.

I will presume you are working in the test data directory, with a subdirectory named `source_data` into which you have copied at least one fits file. For these examples, we assume for convienience your file is named `test.fits`:

```
cd ~
mkdir test_data
cd test_data
mkdir source_data
cp <somepath>/test.fits source_data
```

We'll assume you are working in this directory for the rest of the example. To see if the types from `astrodata_Sample` are discovered, type:

```
typewalk -c
```

This will generate output like the following:

```
directory: . (/home/dpd/test_data)
 test.fits ....................... (CAL) (GEMINI) (GEMINI_NORTH) (GMOS)
 ................................. (GMOS_CAL) (GMOS_IMAGE)
 ................................. (GMOS_IMAGE_FLAT) (GMOS_N) (GMOS_RAW)
 ................................. (IMAGE) (MARKED) (OBSERVED) (RAW)
 ................................. (UNPREPARED)
```

A line should show up for test.fits and any other fits files in the current directory **and** any subdirectory listing the AstroData types which apply to the dataset. The list will contain some Gemini types, such as RAW and UNPREPARED, and if the data in question is Gemini data, types associated with the instrument-mode and processing status.

However, it should also include two types from the sample configuration, `UNMARKED` (or possibly `MARKED` if the dataset has been manipulated by the Sample package previously), and `OBSERVED`.

4.3 Creating An AstroDataType

AstroData types are defined in python class definitions located in the subdirectories of either of two path locations in the our configuration package (`astrodata_Sample`).

- `astrodata_Sample/classification/types` - for typological types
- `astrodata_Sample/classification/status` - for types related to processing pstatus.

The type definition syntax are equivalent, the distinction is only for organization between two sorts of dataset classfication:

1. Classifications that characterize instrument-modes or generic *types* of dataset.
2. Classifications that characterize the processing state of data.

For example, from the `astrodata_Gemini` configuration, the `RAW` and `PREPARED` are "processing types" in `astrodata_Gemini/status/...`, whereas `NICI`, `GMOS` and `GMOS_IMAGE` are "typological types" located in the `astrodata_Gemini/status/...` subdirectory directory.

Since we don't know anything about the instrument or mode that this custom package is being developed for, the sample package will add some somewhat artificial example types as processing types, provided as examples in the sample package that will demonstrate the point in general. For more complicated examples of type requirements, we'll use examples from `astrodata_Gemini`..

To inspect the types in the custom package change directory to `astrodata_Sample/classifications/status` and get a directory listing:

```
cd <base_path>/astrodata_Sample/classifications/status
cat adtype.UNMARKED.py
```

The contents of the file should be as below:

```
1  class UNMARKED(DataClassification):
2      name="UNMARKED"
3      usage = "Processing Type for data not yet 'marked'."
4      parent = "OBSERVED"
5      requirement = PHU({"{prohibit}THEMARK":'.*'})
6
7  newtypes.append(UNMARKED())
```

Note that type source files are read into memory and executed in a prepared environment. Thus there is no need to import the `DataClassification` class from the particular astrodata module, this standard base class is already in scope.

The two elements are the class itself and the `newtypes.append(UNMARKED())` line which instantiates an object of the class and appends it to a list that the ClassificationLibrary can use to inspect datasets. The ClassificationLibrary uses the `newtypes` list to recieve types defined in the module, allowing multiple types to be added to this list in a single type module if desired. At Gemini we have decided to have just one type definition per python type file.

4.3.1 The Class Definition Line by Line

1. `class UNMARKED(DataClassification)`: By convention, we name the class identically to the chosen string name, in this case `UNMARKED`, however this is not required by the system.
2. `name="UNMARKED"`: The classification `name` property stores the string used by the system to identify the type. NOTE: when using type functionality, the user never sees the classification object, and deals with types as strings.
3. `usage="Processing Type for data not yet 'marked'."`: This is used in automatically generated documentation.
4. `parent="OBSERVED"`: This is the type name of a parent class. Note, the type need not also be recognizes as the parent type. The parent member is used to determine overriding assignments in the type tree such that, of course, leaf nodes override root nodes, e.g. for descriptor calculator and primitive set assignments.
5. `requirement = PHU({"{prohibit}THEMARK":'.*'})`: The requirement member uses requirement classes (see below) to define the given type. In this case, this is a PHU check to ensure that "THEMARK" is not set at all in the PHU.
6. `newtypes.append(UNMARKED())`: This line appends an object instance of the new class to a pre-defined `newtypes` array variable. Note, this name is the **class name** from line 1, not the type name, though by convention in Gemini AstroData Types we use the type name as the class name.

4.3.2 The Requirement Classes

The requirement member of a type classification is intended to be declared with an expression built from requirement classes. Again, the type definition is evaluated in a controlled environment and these classes, as well as aliases for convienience, are already in scope.

Concrete Requirements

Concrete Requirments are those that make actual physical checks of dataset characteristic.

Requirement Type	Alias	Description
ClassReq	ISCLASS	For ensuring this type is also some other classification
PhuReq	PHU	Checks a PHU key/value header against a regular expression.

Object Oriented design enables us to extend requirement class ability and/or create new requirements. Examples: the current PHU requirement checks values only against regular expressions, it could be expanded to make numerical comparisons (e.g. to have a dataset type dependent on seeing thresholds). Another example that we anticipate needing is a requirement class that checkes header values in extensions.

Note that currently all type checking resolves to PHU checks, see below for a description of the PHU requirement object.

ISCLASS(other_class_name)

The ISCLASS requirement accepts a string name and will cause the classification to check if the other type applies. Circular definitions are possible and the configuration author must ensure such do not exit.

ISCLASS example:

```
class GMOS(DataClassification):
    name="GMOS"
    usage = '''
        Applies to all data from either GMOS-North or GMOS-South instruments in any mode.
        '''
    parent = "GEMINI"
    requirement = ISCLASS("GMOS_N") | ISCLASS("GMOS_S")

    # equivalent to...
    #   requirement = OR(
    #                     ClassReq("GMOS_N"),
    #                     ClassReq("GMOS_S")
    #                    )

newtypes.append( GMOS())
```

Since there are in fact two GMOS instruments at Gemini, one in Hawaii, one in Chile, the GMOS type really means checking that one of these two instruments was used.

Note: This is also an example of use of the OR requirement, and specifically a convienience feature allowing the "|" symbol to be used for pair-wise or-ing. The included comment shows another form using the OR object constructor which allows more than two operands to be listed.

PHU(keyname=re_val, [keyname2=re_val2 [...]])

The PHU requirement accepts any number of arguments. Each argument name is used as the PHU key name, and the value is a regular expression against which the header value will be compared.

An example:

```
class GMOS_NODANDSHUFFLE(DataClassification):
    name="GMOS_NODANDSHUFFLE"
    usage = "Applies to data from a GMOS instrument in Nod-And-Shuffle mode"
    parent = "GMOS"
    requirement = PHU(NODPIX='.*')

newtypes.append(GMOS_IFU())
```

It is also possible to probit a match, and to use regular expressions for key matching using a special syntax for the key name. This is done by prepending an instruction to the key name, but also requires passing arguments to the

PHU object constructor in a different way. For example the following requirement checks to ensure that the PHU key MASKNAME *does not* match "IFU*":

```
PHU({"{prohibit}MASKNAME": "IFU*"})
```

Note that in this case the arguments are passed to the PHU object constructor as a dictionary. The keys in the dictionary are used to match PHU keys, and the values are regular expressions which will be compared to PHU values.

Generally, python helps instantiating the PHU object by turning the constructor parameter names and their settings into the keys and values of the dictionary it uses internally. However, python doesn't like special characters like "{" in argument names, so to use the extended key syntax requires passing the dictionary.

To use regular expressions in key names (which is also considered dangerous and prone to inefficiency), use the following syntax:

```
class PREPARED(DataClassification):

    name="PREPARED"
    usage = 'Applies to all "prepared" data.'
    parent = "UNPREPARED"
    requirement = PHU( {'{re}.*?PREPARE': ".*?" })

newtypes.append(PREPARED())
```

Due to our legacy reduction software conventions, Gemini datasets which have been run through the system will have a keyword of the sort "<x>PREPARE" with a value set to a time stamp. The need for caution is due to, one, efficiency, since the classification must cycle through all headers to see if the regular expression matches, and two, this technique is prone to a name collision, i.e. in our example above... if a fits PHU happens to have a key matching "*PREPARE" for some other reason than having been processed by the Gemini Package.

Please use this feature with caution.

Logical Requirement Classes

The logical requirement classes use OO design to behave like requirement operators, returning true or false based on a combination of requirements given as arguments.

Requirement Type	Alias	Description
AndReq	AND	For comparing two other requirements with a logical `and`
NotReq	NOT	For negating the truth value of another requirement
OrReq	OR	For comparing two other requirements with a logical `or`

AND(<requirement>,<requirement> [, <requirement> [, <requirement>] ..])

The AND requirement accepts other requirements as arguments. At least two arguments are needed for the AND to be sensible, but if more are present they are also checked for truth value.

It is possible also to use the "&" operator as a logical "and":

```
requirement = AND(PHU("key1", "val1"), PHU("key2", "val2"))
```

...is equivalent to:

```
requirement = PHU("key1", "val1") & PHU("key2", "val2")
```

NOT(<requirement>)

The NOT requirement accepts a single other requirement as arguments. "NOT" is used to negate some requirement. For example at Gemini we do not view a GMOS_BIAS as a GMOS_IMAGE, but it does satisfy the requirements of GMOS_IMAGE. The need for a separate type is due to the fact that GMOS_IMAGE and GMOS_BIAS require different automated reduction (e.g. in a pipeline deployment). To accomplish this we add a `NOT` requirement to GMOS_IMAGE:

```
class GMOS_IMAGE(DataClassification):
    name="GMOS_IMAGE"
    usage = """
        Applies to all imaging datasets from the GMOS instruments
        """
    parent = "GMOS"
    requirement = AND([  ISCLASS("GMOS"),
                         PHU(GRATING="MIRROR"),
                         NOT(ISCLASS("GMOS_BIAS"))  ])

newtypes.append(GMOS_IMAGE())
```

OR(<requirement>,<requirement> [, <requirement> [, <requirement>] ..])

The OR requirement accepts other requirements as arguments. At least two arguments are needed for the OR to be sensible, but if more are present they are also checked for truth value.

It is possible also to use the "|" operator as a logical "or":

```
requirement = OR(PHU("key1", "val1"), PHU("key2", "val2"))
```

...is equivalent to:

```
requirement = PHU("key1", "val1") | PHU("key2", "val2")
```

4.4 Creating a New Descriptor

The descriptor implementations are defined in the `astrodata_Sample/ADCONFIG_Sample/descriptors` directory tree. A descriptor configuration requires the following elements:

1. There must be a "Calculator" object in which the descriptor function must be defined (as a method).
2. The Calculator class must appear in a "calculator index", which are any files in the `descriptors` directory tree named `calculatorIndex.<whatever>.py` where `<whatever>` can be any unique name.
3. The descriptor must be listed in the `DescriptorsList.py` file.

4.4.1 The Calculator Class

The Calculator Class in the Sample package is in the file, `OBSERVED_Descriptors.py`, located in the `descriptors` subdirectory of the `ADCONFIG_Sample` of the `astrodata_Sample` package. It contains just one example descriptor function, `observatory` which relies on the standard MEF PHU key, `OBSERVAT`. Full source:

```
class OBSERVED_DescriptorCalc:
    def observatory(self, dataset, **args):
        return dataset.get_phu_key_value("OBSERVAT")
```

In order for this function to be called for the right type of data, this class must appear in a "calculator index".

4.4.2 The Calculator Index

The Calculator Index for astrodata_Sample is located in the file, `calculatorIndex.Sample.py`, in the `descriptors` subdirectory of `ADCONFIG_Sample` in the `astrodata_Sample` configuration package.

Here is the source:

```
calculatorIndex = {
    "OBSERVED":"OBSERVED_Descriptors.OBSERVED_DescriptorCalc()",
    }
```

Note, the sample index also contains detailed instructions about the format but for our purposes the index should be clear enough. The dictionary key is the string name of a defined AstroData Type, and the value is the class name, including the module it is defined in. The system will parse this name and import the `OBSERVED_Descriptors` module, then store the class in a calculator dictionary.

4.4.3 The DescriptorList.py

The `DescriptorList.py` file contains a list of descriptors definitions. The entries declared must at least declare the name of the new descriptor function. The infrastructure will use these names to create a bridge between AstroData instances and the type-specific descriptor functions.

Adding a New Descriptor to the configuration involves:

1. Adding a "DescriptorDescriptor" to the DescriptorList.py file.
2. Adding the descriptor function to the appropriate Descriptor Calculator class.

The DescriptorList.py File

The contents of `DescriptorList.py` is a list of "DD" object constructors, as follows from the astrodata_Sample package:

```
[
  DD("observatory"),
]
```

To add a descriptor named "telescope" we'd add the following line to the `DescriptorList.py` file:

```
DD("telescope")
```

This tells the infrastructure the name of the descriptor, and in more complicated cases can provide other descriptor metadata to the infrastructure. The final file would look as follows:

```
[
  DD("observatory"),
  DD("telescope")
 ]
```

Adding the Descriptor Function to the CalculatorClass

To add the descriptor once the descriptor is present in the `DescriptorList.py` one merely needs to add a function to the appropriate DescriptorCalculator class. The contents of `OBSERVED_Descriptors.py` module in the astrodata_Sample configuration is:

```
class OBSERVED_DescriptorCalc:
    def observatory(self, dataset, **args):
        return dataset.get_phu_key_value("OBSERVAT")
```

To add the "telescope" descriptor means adding another function to this class:

```
def telescope(self, dataset, **args):
    return dataset.get_phu_key_value("TELESCOP")
```

Note, all descriptors should have the same function signature, including `self`, a `dataset` argument and `**args` to catch all named arguments. The latter is required by the infrastructure so that unexpected parameters can be sent to all descriptor algorithms, some of which may be handled by the infrastructure on behalf of the descriptor function.

4.5 Creating Recipes and Primitive

Primitives are basic transformations. Since different dataset types will sometimes require different concrete implementations of code to perform the given step, the primitive names are shared system-wide, with type-specific implementations.

A "recipe" is a text file containing one primitives (or other recipe) per line. It is thus a sequential view of a reduction or data analysis process. It contains no branching explicitly, but since primitives can be implemented for particular dataset types, there is implicit branching based on dataset type.

4.5.1 Understanding Primitives

Primitives are bundled together in type-specific batches. Thus, for our Sample types of `OBSERVED`, `MARKED`, and `UNMARKED`, each would have its own primitive set. Generally, any given dataset must have exactly one appropriate primitive set per package, which is resolved through the `parent` member of the AstroDataType. Leaf node primitive set assignments override parent assignments.

Which primitive set will be loaded for a given type is specified in index files. Index files and primitive sets must appear in `astrodata_Sample/RECIPES_Sample`, or any subdirectory of this directory. Any arrangement of files into subdirectories below this directory is acceptable. However, by convention Gemini put all "primitive set" modules in the `primitives` subdirectory and put only recipes in this top directory. As the library of recipies grows, Gemini recipes will similarly move to a `recipies` subdirectory, with further subdirectories organized by scientific function.

The astrodata package essentially flattens these directories so moving files around does not affect the configuration or require changing the content of any files, with the exception that the primitive parameter file must appear in the same location as the primitive set module itself.

Primitives Indeces

The astrodata package recursing a `RECIPES_XYZ` directory will look at each filename and if it matches the primitives-index naming convention, `primitivesIndex.<unique_name>.py`, it will try to load the contents of this file and add it to the internal primitive set index. Below is an example of a primitive index file which contributes to the central index:

```
localPrimitiveIndex = {
    "OBSERVED":  ("primitives_OBSERVED.py", "OBSERVEDPrimitives"),
    "UNMARKED":  ("primitives_UNMARKED.py", "UNMARKEDPrimitives"),
    "MARKED"  :  ("primitives_MARKED.py", "MARKEDPrimitives"),
    }
```

The dictionary in the file must be named "localPrimitiveIndex". The key is the type name and the value is a tuple containing the primitives' module basename and the name of the class inside the file, respectively, as strings. These are given as strings because they are only evaluated into python objects if needed.

Note: you can have multiple primitive indeces. As mentioned each index file merely updates a central index collected from all installed packages. The index used in the end is the union of all indices.

Within the sample primitive set, `primitives_OBSERVED.py`, you will find something like the following (notice the Sample may have changed since construction of the document):

```
from astrodata.ReductionObjects import PrimitiveSet

    class OBSERVEDPrimitives(PrimitiveSet):
        astrotype = "OBSERVED"

        def init(self, rc):
            print "OBSERVEDPrimitives.init(rc)"
            return

        def typeSpecificPrimitive(self, rc):
            print "OBSERVEDPrimitives::typeSpecificPrimitive()"

        def mark(self, rc):
            for ad in rc.get_inputs_as_astrodata():
                if ad.is_type("MARKED"):
                    print "OBSERVEDPrimitives::mark(%s) already marked" % ad.filename
                else:
                    ad.phu_set_key_value("THEMARK", "TRUE")
            yield rc

        def unmark(self, rc):
            for ad in rc.get_inputs_as_astrodata():
                if ad.is_type("UNMARKED"):
                    print "OBSERVEDPrimitives::unmark(%s) not marked" % ad.filename
                else:
                    ad.phu_set_key_value("THEMARK", None)
            yield rc
```

Adding another primitive is merely a matter of adding another function to this class. No other index needs to change since it is the primitive set class itself, not the primitives, that are registered in the index. However, note that primitives are implemented with "generator" functions. This type of functions is a standard Python feature. For purposes of writing a primitive all you need to understand about generators is that instead of a''return'' statement, you will use `yield`. Like `return` statement the `yeild` statement accepts a value, and as with "returning a value" a generator "yeilds a value". For primitives this value must be the reduction context passed in to the primitive.

A generator can have many yield statements. The `yield` gives temporary control to the infrastructure, and when the infrastructure is done processing any outstanding duties, execution of the primitive resumes directly after the `yield` statement. To the primitive author it is as if the yield is a `pass` statement, except that the infrastructure may process requests made by the primitive prior to the `yield`, such as a calibration request.

4.5.2 Recipies

Recipes should appear in the RECIPES_<XYZ> subdirectory, and have the naming convention recipe.<whatever>. A simple recipe using the sample primitives is:

```
showInputs(showTypes=True)
mark
typeSpecificPrimitive
showInputs(showTypes=True)
unmark
typeSpecificPrimitive
showInputs(showTypes=True)
```

With this file, named recipe.markUnmark, in the RECIPIES_Sample directory in your test data directory you can execute this recipe with the reduce command:

```
reduce -r markUnmark test.fits
```

The showInputs primitive is a standard primitive, and the argument showTypes tells the primitive to display type information so we can see the affect of the sample primitives. The typeSpecificPrimitive is a sample primitive with different implementations for "MARKED" and "UNMARKED", which prints a message to demonstrate which implementation has been executed.

CHAPTER

FIVE

CONCEPTS

5.1 Background

5.1.1 Dataset Abstraction

The AstroData class traces back to a request by Gemini Astronomers to "handle MEFs better" in our reduction package. A "MEF" is of course a "Multiple-Extension FITS File" and is also Gemini's standard dataset storage format. Investigation showed that the MEF libraries were sufficient for handling "MEFs" as such and the real meaning of the request was for a better dataset abstraction for Gemini's datasets. Gemini MEFs, and MEFs in general, are usually meant to be coherent collections of data; the separate pixel arrays, or extensions, are collocated in a common MEF for that reason. The MEF abstraction itself does not recognize these connections, however, and views the MEF as a list of separate header-data units, their only relation being collocation in the list. Even the PHU, which has certain artifacts as a special header, generally not having pixel data, and which is used as a file-wide header, is merely presented as the header-data unit at index 0. AstroData relies on one pair of relational meta-data available in MEF which is indexing of the list of datasets with (EXTNAME, EXTVER) tuple. EXTNAME operates as an extension-type specifier, and EXTVER serves to associate the extention with other extensions (e.g. by convention ("VAR",2) is the variance plane for ("SCI", 2).

FITS libraries (e.g. pyfits) return opened MEFs as objects which act as lists of Header-Data Units, aka "extensions". AstroData on the other hand is designed to be configured to recognize many internal connections that MEF does not directly encode. AstroData detects the type of the data, and then can make sound assumptions about what the data is and how to handle it. Any particular (python-level) actions on the data are then performed by implementations in the configuration space.

5.1.2 Meta Data

An additional role of the AstroData abstraction is to standardize access to metadata. FITS allows copious metadata in each extension and in the shared "zero'th" extension (aka "the PHU"), but it standardizes only a small subset of what sort of information is stored there. Many properties which are for Gemini essentially universal properties for all of our datasets, across instruments and modes, are not standardized by FITS. For different instruments and modes these bits of information are distributed across different header key-value pairs and stored in different units. This leads to a situation where there is information that is in principle available in all datasets, but which requires instrument-mode-specific coding to be retrieved in a particular unit and with a particular technical meaning. AstroData hides the particulars by allowing functions that calculate the metadata to be defined in the same configuration space in which the dataset type itself is defined.

The AstroDataType system is able to look at any aspect of the dataset to judge if it belongs in a given classification, but the intent is to find characteristics in the MEF's PHU. Using this knowledge, AstroData loads and applies particular instrument-mode-specific methods to obtain general behavior through a uniform interface, as desired for the user. This

uniform interface can be presented not only in the case of meta-data but also in the case of transformations and any other dataset-type-specific behavior.

5.2 =AstroData Types

To first order, Astrodata Types map to instrument-modes, and these provide a good concrete image of what Astrodata Type are. However more abstract types of dataset identification are also possible and make themselves useful, such as generic types such as "IFU" vs "IMAGE", or processing status types such as "RAW" vs "PREPARED".

5.2.1 Dataset Transformations

The Astrodata package's "Recipe System" handles all abstractions involved in transforming a dataset and is built on top of the AstroData dataset abstraction. The system is called the "recipe system" because the top level instructions for transforming data are "recipes", text files of sequential instructions to perform. For example the recipe "overscanCorrect" contains the following (comments removed):

```
prepare
overscanCorrect
addVARDQ
setStackable
averageCombine
storeProcessedBias(clob=True)
```

Each of these instructions is either a "primitive", which is a python function implemented in the configuration space for a dataset of the given classification, or another recipe. Note that the "storeProcessedBias" primitive above takes an argument in this example, "clob(ber)" equals "True", which tells the storage primitive to overwrite (clobber) any pre-existing bias of the same name.

Zero Recipe System Overhead for AstroData-only Users

Use of AstroData does NOT lead to importing any part of the "Recipe System". Thus there there is no overhead borne by users of the AstroData dataset abstract if they do not specifically invoke the Recipe System. Neither the configuration package nor even the related astrodata package modules are imported until the Recipe System is explicitly invoked by the calling program.

5.2.2 The Astrodata Lexicon and Configurations

An Astrodata Configuration package, defining types, metadata, and transformations, relies on a configuration which defines a lexicon of elements which are implemented in the configuration package in a way such that Astrodata can load and apply the functionality involved. Put simply a combination of location and naming conventions allows the configuration author to define elements in a way that astrodata will disover. In the current system there are three types of elements to be concerned with:

- dataset classification names, aka **Astrodata Types**
- high level metadata names, aka **Astrodata Descriptors**
- scientifically meaningful discrete dataset transformation names, aka **Primitives**

Each of these have associated actions:

- **Astrodata Type**: checks a dataset for adherence to an AstroData type classification criteria, generally by checking key-value pairs in the PHU.

- **Astrodata Descriptors**: calculate and return a named piece of high-level metadata for a particular Astrodata Type in particular units.
- **Primitives**: performs a named transformation on a dataset of a particular Astrodata Type.

The "astrodata_Gemini" package contains these definitions for Gemini datasets separated into two parts, one for the basic AstroData related configuration information, and another for Recipe System configuration. The first section, in its own subdirectory in the configuration package directory, in Gemini's case is found in the ADCONFIG_Gemini configuration subdirectory. Configurations in this subdirectory define types, descriptor functions, and other AstroData- related features. The second section, in a sibling subdirectory in the configuration package, in Gemini's case, "RECIPES_Gemini", defines configurations and implementations needed by the Recipe System, such as recipes and primitives.

5.3 Astrodata Type

An Astrodata Type is a named set of dataset characteristics.

Lack of a central system for type detection in our legacy package meant that scripts and tasks in that system make extended checks on the header data in the datasets they manipulate. Often these checks merely verify that the right type of data is being worked on, a very common task, yet these checks can still be somewhat complex and brittle, for example relying on specific headers which may change when an instrument is upgraded.

Astrodata's classification system on the other hand allows defining dataset classifications via configuration packaging such that the type definitions are shared throughout the system. The calling code can refer to type information by a string name for the type, and any subtleties in or changes to the means of detection are centralized, providing some forward and backward compatibility. The system also allows programmers to check dataset types with a single line of code:

```
1  from astrodata.AstroData import AstroData
2
3  ad = AstroData("N20091027S0134.fits")
4
5  if ad.isType("GMOS_IMAGE"):
6     gmos_specific_function(ad)
7
8  if ad.isType("RAW") == False:
9     print "Dataset is not RAW data, already processed."
10 else:
11    handle_raw_dataset(ad)
```

The *isType(..)* function on lines 5 and 8 above is an example of one- line type checking. The one-line check replaces a larger set of PHU header checks which would otherwise have to be used. Users benefit in a forward-compatible way from any future improvements to the named type, such as better checks or incorporation of new instruments and modes, and also gain additional sophistication such as type-hierarchy relationships which are simply not present with the legacy approach.

The most general of benefits to a clean type system is the ability to assign type-specific behaviors and still provide the using programmer with a consistent interface to the type of functionality involved.

5.4 Astrodata Descriptors

A descriptor is named metadata.

It goes without saying that our scientific datasets contain (and require) copious metadata. Significant amounts of "information about the information" is present along with the pixel data regarding an observation and much of it is

important to data analysis processes. The MEF file structure supports such meta-data in the header units of the primary and other extension HDUs.

At first blush the problem retrieving metadata consistently is that while the values of interest are stored in some form in the headers, the header key names do not follow consistent conventions over all. It's easy to assume that there is a one to one relationship between particular metadata headers of different instrument-modes and that the discrepancy is that the developers have merely chosen different header key names. If that were the entire problem a table oriented approach could be used and one could look up the proper header key name for a particular named piece of metadata based on the type of dataset. This particular key would be used to look up the information in the headers.

However, this table driven approach is not workable because the situation turns out to be more complex. Firstly, the units of the given header value may be different for different instruments and modes. A table could be expanded to have columns for the value's storage and return type, but expanding the table in this way would also still not be sufficient for the general case.

The decisive complications that preclude a simple table look-up approach are two, and lead us to a function-based approach. One, the information needed to provide the named metadata is sometimes distributed across multiple key/header values. These require combination or computation, and for different instruments and modes the distribution and combination requires differ. Two, a correct calculation of the metadata sometimes requires use of look-up tables that must be loaded from a configuration space with instrument- specific information, based on the dataset's Astrodata Type.

For metadata which complies with the more simple expectations first considered, widely shared descriptors for some metadata are standard functions able to lookup the meta-data based on standard names or using simple rules that generalize whatever variation there is in the storage of that particular meta-data across different instruments. While it is possible for a descriptor to store its calculated value in the header of the dataset, and return that if called again, essentially caching the value in the header, Gemini descriptors choose as a matter of policy to always recalculate, and leave such caching- schemes to the calling program.

A complete descriptor definition includes the proper unit for the descriptor and a conceptual description (http://gdpsg.wikis- internal.gemini.edu). E.g. Any CCD based data will have an associated "gain", relating to the electronics used to take the image. Given an AstroData instance, ad , to get the "gain" for any supported Astrodata Type, you would use the following source code regardless of the instrument-mode of the dataset:

```
gain = ad.gain()
```

Because the proper descriptors are assigned to the correct Astrodata Types for Gemini Instruments, the line above will take into account any type-specific peculiarities that exist between any supported dataset. The current ADCONFIG_Gemini configuration implementation has descriptors present for all Gemini instruments. See "Gemini AstroData Type Reference" (http://gdpsg.wikis- internal.gemini.edu/index.php?title=UserDocumentation) for a list of available descriptors for Gemini data. Note that descriptor names themselves are not covered in the Astrodata Users Manual itself because they are part of the type-specific configuration package.

5.5 Recipe System Primitives

A primitive is a named transformation.

A primitive is meant to name an abstract dataset transformation for which we will want to assign concrete implementations on a per Astrodata Type basis. E.g. "subtractSky" is a transformation that has meaning for a variety of wavelength regimes which involve subtracting sky frames from the science pixels. Nevertheless, different instruments in different modes will require different implementations for this transformation, due both to differences in the data type and data layout produced by a particular instrument-mode, and also due to different common reduction practices in different wavelength regimes.

Recipe and primitive names both play a role bridging the gap between what the computer does and what the science user expects to be done. The primitives are meant to be human-recognizable steps such as come up in a discussion

among science users about data flow procedures. The recipes are, loosely, the names of data processing work, and the primitives are names for human-recognizable steps in that process. This puts a constraint on how functionally fine grained primitives should becomes. For example at Gemini we have assumed the concept of primitives as "scientifically meaningful" steps means the data should never be in an incoherent or invalid state, scientifically, after a given step. Each step is at least a mini-milestone in a reduction process. So, for example, no primitive should require another primitive to be run subsequent in order to complete its own transformation, and primitives should always output valid, coherent datasets. E.g. there should not be a primitive that modifies pixel data which is followed by a primitive which modifies the header to reflect the change, and instead both steps should be within such a primitive so the data is never reported to the system in an invalid or misleading state.

The fact that recipes can call recipes allows refactoring between recipes and primitives as the set of transformation evolves. A recipe called by a higher level recipe is seen as an atomic step at the level of the calling recipe, and satisfies the requirement. But to experts in the mode being processed, this recipe in turn is made of coherent steps and these steps also satisfy the requirement. Coherent steps which can be broken down into smaller coherent steps are thus probably best addressed with a recipe calling a recipe. This formation helps recipes to work for more types. At bottom primitives have to be executed so that actual python can run and manipulate the dataset, but below a certain level of granularity primitives become inappropriate. Such code, insofar as it is reusable and/or needs to be encapsulated, is written as functions in utility libraries, such as the Gemini "gempy" package.

Formalizing the transformation concept allows us to refactor our data reduction approaches due to unforeseen complications, new information, new instruments, and so on, without having to necessarily change recipes that call these transformations, or the named transformations which the recipes themselves represent. Recipes for specific nodes in the Astrodata Type tree can also be assigned as needed, and the fact that recipes and primitives can be used by name interchangeably ensures that transformations can be refactored and solved with different levels of recipe and primitive as experience grows and needs evolve. This flexibility helps us expand and improve the available transformations while still providing a stable interface to the user.

AstroData is intended to be useful for general python scripting, that is, one does not have to write code in the form of primitives to use Astrodata. And, as mentioned previously, the Recipe System is not automatically imported (i.e. as a result of "import astrodata") so that no overhead is borne by the AstroData user not making use of automation features, such as when writing a script. Further, a script using AstroData benefits from the type, descriptor, validation, and other built in data handling features of AstroData. However, such scripts do not lend themselves to use in a well-controlled automated system, and thus the Recipe System is provided for when there is need for such a system in which to execute the transformation, as with the Gemini Pipeline projects. Unconstrained python scripts lack a consistent control and parameter interface.

When writing primitives all inputs are provided through the Reduction Context, and depending on the control system these may come from the unix command line, the pyraf command line, from a pipeline control system or other software, or by the calling recipes and primitives. Primitive functions are written as python generators, allowing the control system to perform some tasks for the primitive, such as history keeping and logging, keeping lists of stackable images, retrieving appropriate calibrations, and reporting image statistics to a central database, etc., when the primitive "yields".

The automation system is designed to support a range of automation, from a "dataset by dataset, fully automated" mode for pipeline processing of data as it comes from the telescope, through to "interactive automation" where the user decides at what level to initiate automation and where to intervene.

As users advance it may be of interest to consider that strictly speaking primitives transform the `ReductionContext` object and not specifically (or merely) the input datasets. This context contains references to all objects and datasets which are part of the reduction, including the input dataset. While nearly all primitives will access their input datasets and most will modify the datasets and report them as outputs to the reduction context, some primitives may calculate statistics and report these to the reduction context without reporting outputs. In this case the stream inputs will be propagated as inputs to the subsequent primitive. It is the Reduction Context as a whole that is passed into the primitives as the standard and sole argument (besides self) for the primitive. The reduction context must be left in a coherent state upon exit from a primitive.

Below is a prototype recipe in use in our development environment for testing. It performs some initial processing on

RAW data.

```
prepare
overscanSub
overscanTrim
biasSub
flatField
findshiftsAndCombine
```

Presume the above is a generic recipe. This means, given that primitive sets for GMOS_IMAGE, NIRI_IMAGE, etc, implement the named primitives in the recipe, then when the recipe system executes a line such as biasSub , it will execute the "biasSub" member of the appropries PrimitiveSet associate with that type. Thus, if prepare can be implemented for both types, while biassub requires GMOS and NIRI- specific implementations, then "prepare" can be implemented as a shared recipe or in the GEMINI primitive set, while those that require special implementation are implemented in the appropriate GMOS or NIRI primitive sets within the correct part of the configuration.

5.5.1 Some Benefits of the Primitive Concept

Use of primitives instead of scripts for reduction processes has a major side benefit besides enjoying automation features supplied by the Recipe System. This benefit is due to the fact that the concept of the primitive as a named transformation is bound to the spoken language that Instrument Scientists, PIs, data analysts and the data software group at Gemini use to discuss data flow procedures. This crossover between terms in our formal system and in our less formal spoken language has promoted consistency between the two. For example, when breaking reductions down into discrete chunks which can be implemented and shared when possible the process helps us understand what truly differentiates different implementations of the same named transformation. Sharing of code not only saves developers the effort of reimplementation, but more importantly it promotes consistency and provides locations in the system where wide ranging changes in policy can be implemented accommodating the inevitable evolution of reduction software.

In short, discussing how to break down Gemini's classical reduction procedures into recipes made of reusable primitives has had the effect of clarifying our understanding of these procedures. Sometimes the responsibilities of tasks in our legacy system had clear boundaries, such as for gemarith , but for other tasks, such as the "prepare" task in each instrument's package, the boundaries of responsibility were less clear. Adapting such transformation concepts which are already in our spoken lexicon to a more structures software environment represented with concrete implementations, guides us to creating a more concrete definition for `prepare` and for for the steps in `prepare`'. A source of these discrepancies is different practices in different wavelength and mode regimes which cause different interpretations of how far data should be reduced from the raw state to a more general starting point appropriate for a "typical PI". Flexibility in the system allows satisfaction of these special needs while developing truly shared transformation concepts.

Natural Emergence of Reusable Primitives

Reusable code naturally emerged from the process above because the work of isolating the steps taken in a data handling process naturally reveals similar or identical steps shared by other processes, which can then easily be implemented at a shared level. In practice, even if creating a recipe which is over-all very instrument and mode-specific, there seem to emerge general purpose steps which can be of benefit in a toolkit of primitives. The corollary to this is that in the future after implementing many of these reusable pieces as part of accomplishing project-specific aims at the time, new project-specific tasks will be able to select from and reuse them freely. However, the original implementor saves time by being able to focus on their task at hand, and make a primitive they hope is reusable, without focusing on that as a requirement. This way we hope to benefit from opportunistic sharing of code via the natural evolution of primitives.

The subsequent author has several options based on the needs of the project at hand:

1. generalize the previous attempt at a general solution to leverage the work already done

2. write a new generalization
3. write a version which is primarily designed to be useful as a primitive in the project's use case

The design of the recipes and primitives of the Recipe System is intended to facilitate negotiating these options in an environment with fall-backs and which does not cement you into a particular layout of your transformations. Option 3 is undesirable in general, given sufficient time and an ideal understanding of the problem, but given deadlines based on real world calendar events like instruments going on sky, commissioning, et cetera, it is a desirable fall-back option because in the end it allows the developer to focuss entirely on the problem at hand if it proves hard to generalize. Option 3 interacts with option 1 insofar as often a developer may find, when attempting to generalize code, that one has created a modified version that does work for the new case, but has broken the old case. The flexibility of the Recipe System allows the developer to split the code, use it as two different primitives each assigned to the correct type of datasets using the type system, allowing one to work toward a project milestone and defer more complete generalization of the primitive.

If time is not given specifically to solve the problem afterwards, then at least when a third author requires the same functionality, they then have the same options and fall-backs, with a greater selection of potential primitives to use or generalize to suit their own purposes, with preservation of old behavior as need be. For Gemini primitives we are attempting to produce robust, general, primitives from the start, but this ability of the Recipe Configuration to evolve is still a crucial aspect of the primitive system.

Test Case at Gemini Observatory: Refactoring Python Scripts into Recipes and Primitives
++++++++++++++++++++++

We (GDPSG and DA teams) have performed the exercise of breaking down a set of pre-existing scripts into recipes and primitives in the case of some instrument monitoring scripts which are set up on a cron job. Separate from the issue of the quality of the code being thus preserved, the procedure for refactoring into the recipe/primitive form turned out relatively easy and to involve the following:

1. Finding where (potential) milestone states of the data occur in the script being refactored. These are places where the dataset and headers are coherent, and any information the reduction context should be informed of has been prepared and is available. Note, some potential milestone states, when considered too fine grained will be bundled together as a single transformation.
2. Naming the source code between each of these milestone states, and identifying its input, output, and specific responsibilities.
3. Cutting and pasting (or re-entering) source from the script into a primitive set class, adding adapter code which fetches or stores information in the reduction context to and from variables the script uses in its legacy form. The code can be largely left as is as primitives are simply python code, so long as input/output is adapted to the reduction context.
4. Writing a recipe is using the steps created above.

Regarding the quality of the code thus being preserved, while it was minimal upon analysis, as is often the case it had the advantage of being deployed and functional. It is the intent of the Recipe system to allow rapid adaptation of code into the system, as well as to enable more intimately and well behaved transformations to be integrated, and for there to be iterative refactoring paths from the former to the latter.

The primitives in the test case were developed into a separate recipe package (not in astrodata_Gemini/RECIPES_Gemini) which is added to the Astrodata package's RECIPEPATH environment variable. As a stand alone package for a particular internal purpose it was not as important for these primitives to follow idealized standards as it is for the general purpose "astrodata_Gemini" package. Thus, instead of formal analysis of the scripts and a resultant design, these primitives were abstracted using the method above, from the ad hoc design of the scripts that had been doing the work.

Even with lack of a formal structure to the refactoring, and the devil-you-know approach to preserving the functioning of the code, the process of adaptation to the recipe/primitive structure provides some natural order and formalism in the process of identifying the de facto transformations in the script. Improvement is incremental and procedes de facto design of the script, (i.e. a potentially ad hoc, design-as- you-go, non-design). But even in this case, at the very least,

the above analysis will lead to a sequential list of the steps in the script. That alone is a good starting point for making a complete replacement if that is necessary. Subsequent work on the recipes and component primitives only improves the exposure of the work, the consciousness of the ordering of operations, and merging of common functionality into common code.

In the case of our instrument monitoring example the result of the refactoring to the Recipe System is functional and in use. The resulting recipes made use of some primitives from the Gemini library of primitives, and could benefit from more refactoring allowing both some primitives from the main package to be used (i.e. the scripts performed, and primitives were adapted around a custom "prepare" step on GMOS data), and also to allow several of the primitives created to be made more robust and moved into the main package.

5.5.2 Recipes calling Recipes

Recipes can in fact call other recipes as well as primitives. Primitives, also, can call recipes and other primitives. During execution, the Astrodata Recipe System makes little distinction between recipes and primitives and from the view of those invoking recipes and primitives, recipe and primitive names are interchangeable. E.g. a user executing recipes through the reduce command line program can just as easilly give a primitive name to the "reduce" command as a "recipe" name, and reduce will execute the primitive correctly. Still the general picture we tend to speak of is one in which we have a top level recipe for standard processes such as making a processed bias, which list the steps that the data must go through to complete the processing named by the recipe. In principle these steps are implemented in python and different types will be associated to different implementation, but again, in reality, the recipe may be calling other recipes which are broken down further into steps of either sub-recipes or final primitives.

It is a judgment call how fine grained the steps in a recipe should be, and this in principle drives how fine grain primitives should be. However, what is appropriate to view in a recipe of a certain name and scope may not be the same granularity level which is appropriate for specialists in the data regime being processed, as the recipe will in general be associated with some general purpose concepts, and should have meaning for someone with general purpose knowledge. Sometimes if the top level recipe were to name every step which an Instrument Analyst or Data Processing Developer found distinct and "scientifically meaningful" this would lead to a too finely grained list of steps, which would obscure the big picture of how the transformation named is executed.

In this case, which is common, then the more finely grained steps should be bundled together into recipes which then are used as single statements in higher level recipes. The ability for recipes to call recipes ensures steps can be named whatever is semantically appropriate for whatever the scope of the transformation named might be. At one extreme the recipe system can support a processing paradigm in pipelines which invokes reduction with the most general instructions, "do the appropriate thing for the next file", and at the other extreme it allows users to decide what to treat as atomic processes and when to intervene.

The fact that primitives (should) always leave datasets at some milestone of processing provides some security for the user that they will not perform an operation that puts the dataset in an incoherent state. Breaking down recipes into sub-recipes and so on into primitives truncates at the lowest level when we have primitives that, however focused, modify the data (or reduction context) in some significant way and leave the dataset at some milestone of reduction, however minor a "milestone" it may be. It's also possible, especially if a primitive is adapted from a script, that a primitive will be monolithic, and cannot be broken down into a recipe until more finely grained primitives are created. The interchangeability of recipes and primitive names is meant to encourage such refactoring, as any reusable set of primitives is considered more useful than a monolithic primitive performing all the functions of the reusable set at once.

5.6 AstroData Lexicon

A lexicon is a list of words, and this is what the designer of an Astrodata configuration creates. The set of terms adhere to a grammar (types of elements that can be defined) and establishes a vocabulary about dataset types, metadata, and transformations. Firstly, the configurations define string type names, and criteria by which they can be identified as a

given type of dataset. Then they construct names for and describe metadata one expects to be associated with these datasets. Finally they create names for and describe transformations that can be performed on datasets.

Datasets of particular Astrodata Types, sufficiently defined, can thus be recognized by astrodata and the other type-specific behaviors can be assigned. For example, the "astrodata_Gemini" package is the public configuration package defining data taken by Gemini instruments. Descriptors for all instruments have been created, and early implementations of primitives for GMOS_IMAGE and GMOS are available (and under continued development).

For complete documentation of the ADCONFIG_Gemini type and descriptor package see "Gemini AstroData Type Reference", available at http://gdpsg.wikis- internal.gemini.edu/index.php?title=UserDocumentation.

The astrodata package itself has no built-in type or descriptor definitions. It contains only the infrastructure to load such definitions from an astrodata configuration package directory (the path of which must appear in the `PYTHONPATH`, `RECIPEPATH`, or `ADCONFIGPATH` environment variables as a directory complying with the "astrodata_xxx" naming convention, and containing at least one of either `ADCONFIG_<whatever>` or `RECIPES_<whatever>` sub-packages.

Here is an part of the Gemini type hierarchy, the GMOS_IMAGE branch of the GMOS types:

astrodata package using the type definitions and graphviz, please forgive its purely utilitarian nature.

This diagram shows GMOS_IMAGE is a child type of the GMOS type, which in turn is a child of the GEMINI type. The children of GMOS_IMAGE are other types which share some or all common primitives or other properties with GMOS_IMAGE, but which may in some cases require special handling. The diagram shows descriptor calculator and primitive set assignments. A descriptor calculator (a set of descriptor functions) is assigned to GMOS, from which GMOS_IMAGE and GMOS_SPECT inherit the same descriptors as there is nothing more specific assigned.

The graph also shows primitive sets assigned to GEMINI, GMOS, and GMOS_IMAGE. Since a primitive set specific to GMOS_IMAGE is present in the configuration, it would be used for transformations applying to GMOS_IMAGE datasets rather than the GMOS or GEMINI primitives. However the primitive set class for GMOS_IMAGE happens to be defined in astrodata_Gemini as a child class of the GMOS primitive set, and the GMOS primitive set as the child of the GEMINI primitive set, so in fact, the members can be shared unless intentionally overridden.

Primitives associated with the GEMINI Astrodata Type are generally just bookkeeping functions which rely on features of the Recipe System as few pixel transformations can be entirely generalized across all Gemini datasets, though some can. In the future, some of these primitives will be moved to a very general type associated with any MEF for which a more specific type is not recognized.

INDEX

Symbols

A

C

D

E

G

H

I

L

P

R

S

W