AstroData Users Manual

Release X1.0.1

Kathleen Labrie

November 14, 2014

CONTENTS

CHAPTER

ONE

INTRODUCTION

1.1 What is AstroData?

The AstroData class is a tool to represent datasets stored in Multi-Extension FITS (MEF) files. It provides uniform interfaces for working on datasets from different instruments and their observational modes. Configuration packages are used to describe the specific data characteristics, layout, and to store type-specific implementations. Once a MEF has been opened with AstroData, the object it is assigned to knows essential information about itself, like from which instrument this data comes from, how to access the header information, etc.

Multi-extension FITS files are generalized as lists of header-data units (HDU), with key-value pairs populating headers, and pixel values populating data arrays. AstroData interprets a MEF as a single complex entity. The individual "extensions" within the MEF are available with normal Python list ("[]") syntax.

In order to identify types for the dataset and provide type-specific behavior, AstroData relies on configuration packages. A configuration package (eg. `astrodata_Gemini`) contains definitions for all instruments and modes. A configuration package contains type definitions, meta-data functions, information lookup tables, and any other code or information needed to handle specific types of dataset.

1.2 Installing AstroData

The `astrodata` package has several dependencies like `numpy`, `astropy`, and others. The best way to get everything you need is to install Ureka, http://ssb.stsci.edu/ureka/.

WARNING: The Ureka installation script will not set up IRAF for you. You need to do that yourself. Here's how:

```
$ cd ~
$ mkdir iraf
$ cd iraf
$ mkiraf
-- creating a new uparm directory
Terminal types: xgterm,xterm,gterm,vt640,vt100,etc.
Enter terminal type: xgterm
A new LOGIN.CL file has been created in the current directory.
You may wish to review and edit this file to change the defaults.
```

Once this is done, install `gemini_python`. The `astrodata` package is currently distributed as part of the `gemini_python` package. The `gemini_python` package, `gemini_python-X1.tar.gz`, can be obtained from the Gemini website:

> http://www.gemini.edu/sciops/data-and-results/processing-software

1.2.1 Recommended installation

It is recommended to install the software in a location other than the standard python location for modules (the default `site-packages`). This is also the only solution if you do not have write permission to the default `site-packages`. Here is how you install the software somewhere other than the default location:

```
$ python setup.py install --prefix=/your/favorite/location
```

`/your/favorite/location` must already exist. This command will install executable scripts in a `bin` subdirectory, the documentation in a `share` subdirectory, and the modules in a `lib/python2.7/site-packages` subdirectory. The modules being installed are `astrodata`, `astrodata_FITS`, `astrodata_Gemini`, and `gempy`. In this manual, we will not use `gempy`.

Because you are not using the default location, you will need to add two paths to your environment. You might want to add the following to your .cshrc or .bash_profile, or equivalent shell configuration script.

For tcsh:

```
setenv PATH /your/favorite/location/bin:${PATH}
setenv PYTHONPATH /your/favorite/location/lib/python2.7/site-packages:${PYTHONPATH}
```

For bash:

```
export PATH=/your/favorite/location/bin:${PATH}
export PYTHONPATH=/your/favorite/location/lib/python2.7/site-packages:${PYTHONPATH}
```

If you added those lines to your shell configuration script, make sure your `source` the file to activate the new setting.

For tcsh:

```
$ source ~/.cshrc
$ rehash
```

For bash:

```
$ source ~/.bash_profile
```

1.2.2 Easier but more dangerous installation

Assuming that you have installed Ureka and that you have write access to the Ureka directory, this will install `astrodata` in the Ureka `site-packages` directory. WARNING: While easier to install and configure, this will modify your Ureka installation.

```
$ python setup.py install
```

This will also add executables to the Ureka `bin` directory and documentation to the Ureka `share` directory.

With this installation scheme, there is no need to add paths to your environment. However, it is a lot more complicated to remove the Gemini software in case of problems, or if you just want to clean it out after evaluation.

In tcsh, you will need to run `rehash` to pick the new executables written to `bin`.

1.2.3 Smoke test the installation

Just to make there is nothing obviously wrong with the installation and configuration, we recommend that you run the following smoke tests:

```
$ which typewalk

Expected result: /your/favorite/location/bin/typewalk

$ python
>>> from astrodata import AstroData

Expected result: Just a python prompt and no error messages.
```

1.3 AstroData Support

This release of `astrodata` as part of `gemini_python-X1` is an early release of what we are working on. It is not a fully supported product yet. If you do have questions or feedback, please use the Gemini Helpdesk but keep in mind that the ticket will be addressed on a best-effort basis only.

CHAPTER

TWO

MEF INPUT/OUTPUT OPERATIONS AND EXTENSIONS MANIPULATION

In this section, we will show and discuss how to read and write to a Multi-Extension FITS (MEF) file.

A MEF file is a Primary Header Unit (PHU) with a list of Header Data Units (HDU), commonly referred to as extensions. In AstroData, the extension numbering is zero-indexed, the first extension is `myAstroData[0]` where `myAstroData` is an open AstroData object. If you are familiar with IRAF, then keep in mind that in IRAF the extensions are 1-indexed. For example:

```
# In IRAF
display myMEF[1]

# With AstroData
numdisplay.display(myAstroData[0].data)
```

The first data extension is 1 in IRAF, but it is 0 in AstroData. All Python arrays are zero-indexed, so AstroData was made compliant with modern practice.

Try it yourself

If you wish to follow along and try the commands yourself, download the data package, go to the `playground` directory and copy over the necessary files.

```
cd <path>/gemini_python_datapkg-X1/playground
cp ../data_for_ad_user_manual/N20110313S0188.fits .
cp ../data_for_ad_user_manual/N20110316S0321.fits .
cp ../data_for_ad_user_manual/N20111124S0203.fits .
```

Then launch the Python shell:

```
python
```

2.1 Open and access existing MEF files

An AstroData object can be created from the name of the file on disk, a URL, or from PyFITS HDUList or HDU instance. An existing MEF file can be opened as an AstroData object in `readonly`, `update`, or `append` mode. The default is `readonly`.

Here is a very simple example on how to open a file in `readonly` mode, check the structure, and then close it:

```
1 from astrodata import AstroData
2
3 ad = AstroData('N20111124S0203.fits')
```

```
ad.info()
ad.close()
```

The first line, imports the `AstroData` class. The `info` method prints to screen the list of extensions, their name, size, data type, etc. Here's what the example above will output:

```
Filename: N20111124S0203.fits
    Type: AstroData
    Mode: readonly

AD No.    Name          Type      MEF No.  Cards    Dimensions   Format
         hdulist       HDUList
         phu           PrimaryHDU    0       179
         phu.header    Header
[0]      ('SCI', 1)    ImageHDU      1        71    (4608, 1056)  float32
            .header    Header
            .data      ndarray
[1]      ('SCI', 2)    ImageHDU      2        71    (4608, 1056)  float32
            .header    Header
            .data      ndarray
[2]      ('SCI', 3)    ImageHDU      3        71    (4608, 1056)  float32
            .header    Header
            .data      ndarray
[3]      ('SCI', 4)    ImageHDU      4        71    (4608, 1056)  float32
            .header    Header
            .data      ndarray
[4]      ('SCI', 5)    ImageHDU      5        71    (4608, 1056)  float32
            .header    Header
            .data      ndarray
[5]      ('SCI', 6)    ImageHDU      6        71    (4608, 1056)  float32
            .header    Header
            .data      ndarray
```

To open the file in a mode other than `readonly`, one specifies the setting of the `mode` argument:

```
ad = AstroData('N20111124S0203.fits', mode='update')
```

2.1.1 Accessing the content of a MEF file

Conceivably one opens a file to access its content. Manipulations of the pixel data and of the headers are covered in more details in later sections (*Section 5 - Pixel Data* and *Section 4 - FITS Headers*, respectively). Here we show a few very basic examples on how to access pixels and headers.

The pixel data in an AstroData object are stored as Numpy `ndarray`. Any `ndarray` operations are valid. We use Numpy's `average` function in the example below.

The headers in an AstroData object are stored as PyFITS headers. Any PyFITS header operations are valid.

```
from astrodata import AstroData
import numpy as np

ad = AstroData('N20111124S0203.fits')

for extension in ad:
   print 'Extension :', extension.extname(), ',', extension.extver()
   #
   # Access the pixel data
   data = extension.data
```

```
11      print 'data is of : ', type(data)
12      print 'The pixel data type is: ', data.dtype
13      print 'The average of all the pixels is: ', np.average(data)
14      #
15      # Access the header
16      hdr = extension.header
17      print 'The value of NAXIS2 is: ', hdr.get('NAXIS2')
18      print
```

(Python Beginner's Note 1: The ''#'' on line 8 and 14 are not necessary to the code but simplify the cut and paste of the statements from the HTML page to the Python shell, without affecting readability.)

(Python Beginner's Note 2: In the Python shell, when you are done inputing the statements of a loop, you indicate so by typing return to create an empty line. So, after you have written the last ''print'' statement, type return on the ''...'' line, this will launch the execution of the loop.)

Now let us discuss the example.

As stated above, the pixel data are stored in `numpy.ndarray` objects. Therefore, Numpy needs to be imported if any `numpy` operations is to be run on the data. This is done on Line 2, using the standard import convention for Numpy.

On Line 6, the for-loop that will access the extension sequentially is defined. Only the extensions are returned, the Primary Header Unit (PHU) is not sent to the loop. Access to the PHU is discussed in *Section 4 - FITS Headers*.

In an AstroData object, each extension is given, in memory if not on disk, an extension name and an extension version. Line 7 accesses that information.

On lines 10 to 13, the pixel data for the current extension is assigned to the variable `data`, and then the array is explored a bit.

On lines 16 and 17, the header associated with the extension is assigned to the variable `hdr`, and the value for the keyword `NAXIS2` is retrieved.

Note that for both `data` and `hdr`, the pixels or the headers are NOT copied, the new variables simply point to the information stored in the AstroData object. If `data` or `hdr` are modified, the AstroData object itself will be modified.

In the example above, a loop through the extensions is used. To access a specific extension by name, it is also possible to do something like this:

```
data = ad['SCI',1].data
print 'Value of NAXIS2: ', ad['SCI',1].header.get('NAXIS2')
```

or if not using the names, using the positional number for the extension:

```
header = ad[0].header
print 'Extension name and version for extension 0: ', \
   ad[0].extname(), ad[0].extver()
```

Note that the extension positions are zero-indexed, `ad[0]` is not the PHU, it is the first extension.

2.2 Modify Existing MEF Files

To modify an existing MEF file, it must have been opened in the `update` mode. While a MEF opened in any mode can be modified at will in memory, only an file opened in `update` (or `append`) mode can be overwritten on disk.

Here we give examples on how to append an extension, how to insert an extension, and how to remove an extension. Finally we show how to write the updated AstroData object to back disk as a MEF file.

Extension manipulations have been chosen for this discussion, but any other type of modifications, eg. pixel arithmetics, header editing, etc. could have been chosen instead. Extension manipulations are a good introduction to the structure and extension-naming convention of the AstroData MEF file representation.

2.2.1 Opening the files

For the extension manipulation examples of the next subsections, two files are needed, one to serve as the main file to edit, and another from which we can extract extension from for inserting or appending into the first.

```
1 from astrodata import AstroData
2
3 ad = AstroData('N20110313S0188.fits', mode='update')
4 ad.info()
5
6 adread = AstroData('N20110316S0321.fits', mode='readonly')
7 new_extension = adread["SCI",2]
8 new_extension.info()
```

The first step is always to import the `AstroData` class (line 1). On Line 3, the main dataset is open as an AstroData object in `update` mode. The other dataset is open in `readonly` mode (line 6). Note that the `readonly` is optional as this is the default mode.

An extension is "extracted" from the second dataset on line 7. It is important to realize that this does NOT create a copy of the extension. The variable `new_extension` simply *points* to the data stored in `adread`.

The first dataset's structure is (from line 4, `ad.info()`):

```
Filename: N20110313S0188.fits
    Type: AstroData
    Mode: readonly

AD No.    Name          Type       MEF No.  Cards    Dimensions   Format
         hdulist       HDUList
         phu           PrimaryHDU    0       179
         phu.header    Header
[0]      ('SCI', 1)    ImageHDU      1        37    (2304, 1056)  float32
            .header    Header
            .data      ndarray
[1]      ('SCI', 2)    ImageHDU      2        37    (2304, 1056)  float32
            .header    Header
            .data      ndarray
[2]      ('SCI', 3)    ImageHDU      3        37    (2304, 1056)  float32
            .header    Header
            .data      ndarray
```

The structure of the new extension is (from line 8. `new_extension.info()`):

```
Filename: N20110316S0321.fits
    Type: AstroData
    Mode: update

AD No.    Name          Type       MEF No.  Cards    Dimensions   Format
         hdulist       HDUList
         phu           PrimaryHDU    0       147
         phu.header    Header
[0]      ('SCI', 2)    ImageHDU      1        39    (2304, 1056)  float32
            .header    Header
            .data      ndarray
```

2.2.2 Appending an extension

Appending and inserting extension can be tricky. The first difficulty comes from the naming of the extensions. No two extensions are allowed to have the same EXTNAME, EXTVER combination. When appending or inserting, the user must either specify the EXTNAME, EXTVER of the new extension or use the `auto_number` options which tries to do something sensible to keep the MEF structurally valid.

The second difficulty is due to the fact an assignment of an extension to a variable is just a "reference", a "link", it is not a copy. Any changes to the variable standing as a pointer to an extension in an AstroData object will affect the AstroData object itself.

The `append` method adds an extension as the end of a dataset. Here is an example appending an extension to a dataset. Further discussion follows.

```
ad.append(new_extension, auto_number=True, do_deepcopy=True)
ad.info()
```

The AstroData method `append()` is used to append an extension to an AstroData object. On the first line, the extension `new_extension` gets appended to the `ad` dataset.

As you see in the previous subsection, the extension name and extension version number of the extension in `new_extension` is ['SCI', 2]. There is already an extension named and versioned that way in `ad` (see the result of `ad.info()` in the previous subsection). Therefore, to avoid conflict, the argument `auto_number` is set to `True`.

As you can see from the output of `ad.info()` after the `append` call:

```
1  Filename: N20110313S0188.fits
2      Type: AstroData
3      Mode: update
4
5  AD No.    Name          Type       MEF No.  Cards    Dimensions   Format
6          hdulist       HDUList
7          phu           PrimaryHDU    0       179
8          phu.header    Header
9  [0]     ('SCI', 1)    ImageHDU      1        37    (2304, 1056)  float32
10            .header    Header
11            .data      ndarray
12 [1]     ('SCI', 2)    ImageHDU      2        37    (2304, 1056)  float32
13            .header    Header
14            .data      ndarray
15 [2]     ('SCI', 3)    ImageHDU      3        37    (2304, 1056)  float32
16            .header    Header
17            .data      ndarray
18 [3]     ('SCI', 4)    ImageHDU      4        39    (2304, 1056)  float32
19            .header    Header
20            .data      ndarray
```

There is a new extension at the bottom of the list named ['SCI', 4] (lines 18-20). The `auto_number` feature figured out that ['SCI', 1 to 3] already existed and assigned the new ['SCI'] extension a version number of 4 to avoid conflict.

Now, there is a second issue to deal with. Since `new_extension` is just a reference to the extension stored in `adread`, when `auto_number` changes the version number to 4, the extension in `adread` will also be modified, corrupting the source. Moreover, if other modifications are made to the extension inserted in `ad`, it will modify `adread` too. There are times when it will not matter at all, for example if `adread` is scheduled to be closed anyway, but if `adread` is to be used later in the script, it should not be modified like that.

To cut the link between the extension appended to the dataset and the source, the argument `do_deepcopy` is set to `True`. This will often be needed. It is not the default because of memory usage concerns, and to force the user to think before creating copies and decide if it is truly needed.

2.2.3 Inserting an extension

When inserting an extension into an AstroData object, the same caution to extension name and version, and to the reference versus copy issues applies. Instead of repeating ourselves, we refer to users to the discusssion above in the "Appending" section.

With that in mind, let us present a few examples of insertion.

Simple insertion

To insert an extension between the PHU and the first extension:

```
1 new_extension = adread['SCI',1]
2 new_extension.rename_ext('VAR')
3 new_extension.info()
4
5 ad.insert(0, new_extension)
6 ad.info()
```

On Line 2, we rename the extension to 'VAR' simply to make it stand out once inserted. Also, note that since the new extension is named ['VAR',1], it does not conflict with any of the extensions already present in `ad`, therefore there were no need for activating the `auto_number` option. We did not use `do_deepcopy` either. As consequence, the source of `new_extension`, `adread`, has been modified (in memory).

Before and after insertion, `new_extension` has this structure:

```
1  Filename: N20110316S0321.fits
2      Type: AstroData
3      Mode: update
4
5  AD No.    Name          Type       MEF No.  Cards    Dimensions   Format
6          hdulist       HDUList
7          phu           PrimaryHDU    0       147
8          phu.header    Header
9  [0]     ('VAR', 1)    ImageHDU      1        39    (2304, 1056)  float32
10            .header    Header
11            .data      ndarray
```

The actual insertion takes place on Line 5. The syntax requires the first argument to be the position number, or the name and version, of the extension to "push". The new extension will be inserted *before* the extension specified in the statement. The second argument is obviously the extension to insert into `ad`.

After insertion, `ad` looks like this:

```
1  Filename: N20110313S0188.fits
2      Type: AstroData
3      Mode: update
4
5  AD No.    Name          Type       MEF No.  Cards    Dimensions   Format
6          hdulist       HDUList
7          phu           PrimaryHDU    0       179
8          phu.header    Header
9  [0]     ('VAR', 1)    ImageHDU      1        39    (2304, 1056)  float32
10            .header    Header
11            .data      ndarray
12 [1]     ('SCI', 1)    ImageHDU      2        37    (2304, 1056)  float32
13            .header    Header
14            .data      ndarray
15 [2]     ('SCI', 2)    ImageHDU      3        37    (2304, 1056)  float32
```

```
16              .header    Header
17              .data      ndarray
18  [3]      ('SCI', 3)    ImageHDU      4        37    (2304, 1056)  float32
19              .header    Header
20              .data      ndarray
21  [4]      ('SCI', 4)    ImageHDU      5        39    (2304, 1056)  float32
22              .header    Header
23              .data      ndarray
```

As one can see, `new_extension` is now the first extension in the file structure, or at position 0 (Lines 9-11), and the other extensions have been moved "down".

A tricky insertion

The name of the new extension ['VAR',1] was not in conflict with any pre-existing extensions in the original AstroData object. Let us insert the new extension again, in another position in the current AstroData object. This time, there is a ['VAR', 1] in the AstroData object, and `auto_number` and `do_deepcopy` are required.

The `auto_number` option is required to avoid the extension name clash. Less obvious is why `do_deepcopy` is required, assuming that we do not care about the impact on `adread`. The reason is subtle, and clearly illustrate why extension manipulations is probably the most tricky concept in this manual, yet fortunately not that commonly needed.

When we inserted `new_extension` above, we did not use `do_deepcopy`. Therefore, if we were to modify `new_extension`, like through `auto_number`, we would be modifying not only the source, `adread`, but also that extension we have already added to `ad` !

As you can see, it is vitally important to understand was is a true copy and what is a reference to something else when dealing with extensions. Beginners might want to use `do_deepcopy=True` as a default, until they are comfortable with the concept of references. *Beware* however that memory usage can rise significantly.

Here is how one would insert `new_extension` somewhere else in `ad`. In the example, the extension is inserted between the current third and fourth extension. Since position ID are zero-indexed, this means between position 2 and 3.

```
ad.insert(3, new_extension, auto_number=True, do_deepcopy=True)
ad.info()
```

Look at what happened to the name of the newly inserted extension (Lines 18-20):

```
1   Filename: N20110313S0188.fits
2       Type: AstroData
3       Mode: update
4
5   AD No.    Name          Type      MEF No.  Cards    Dimensions   Format
6           hdulist       HDUList
7           phu           PrimaryHDU    0       179
8           phu.header    Header
9   [0]     ('VAR', 1)    ImageHDU      1        39    (2304, 1056)  float32
10              .header    Header
11              .data      ndarray
12  [1]     ('SCI', 1)    ImageHDU      2        37    (2304, 1056)  float32
13              .header    Header
14              .data      ndarray
15  [2]     ('SCI', 2)    ImageHDU      3        37    (2304, 1056)  float32
16              .header    Header
17              .data      ndarray
18  [3]     ('VAR', 5)    ImageHDU      4        39    (2304, 1056)  float32
19              .header    Header
```

```
            .data      ndarray
[4]      ('SCI', 3)    ImageHDU      5        37    (2304, 1056)  float32
            .header    Header
            .data      ndarray
[5]      ('SCI', 4)    ImageHDU      6        39    (2304, 1056)  float32
            .header    Header
            .data      ndarray
```

The automatic renumbering assigned an extension number of 5 to the newly inserted extension. One might have expected that 2 would be assigned as the next available version number of the 'VAR' name. This behavior was designed to prevent the software from making the scientific assumption that the new extension is in anyway associated with another. Normally, it is assumed that all extension with a given EXTVER are scientifically associated. `auto_number` has no way to know which extension is scientifically associated with an other. The purpose of `auto_number` is solely to keep the AstroData structure sound and prevent corruption due to clashing name/version pairs. It is the job of the programmer, who has the scientific knowledge of the associations, to name and version the extensions correctly when that matters.

Using name/version pairs instead of position ID

The position at which the insertion is to take place can be given as the positional ID like in the examples above, or by specify the extension name and version.

```
new_extension = adread['SCI',3]
new_extension.rename_ext('VAR')
ad.insert(('SCI',4), new_extension)
ad.info()
```

In this example, a new extension is retrieved from the source and renamed 'VAR' to avoid name conflict and keep the example simple. The insertion takes place on the third line where the position of insertion is specfied as `('SCI',4)`. The resulting `ad` looks like this:

```
Filename: N20110313S0188.fits
    Type: AstroData
    Mode: update

AD No.    Name          Type      MEF No.  Cards    Dimensions   Format
         hdulist       HDUList
         phu           PrimaryHDU    0       179
         phu.header    Header
[0]      ('VAR', 1)    ImageHDU      1        39    (2304, 1056)  float32
            .header    Header
            .data      ndarray
[1]      ('SCI', 1)    ImageHDU      2        37    (2304, 1056)  float32
            .header    Header
            .data      ndarray
[2]      ('SCI', 2)    ImageHDU      3        37    (2304, 1056)  float32
            .header    Header
            .data      ndarray
[3]      ('VAR', 5)    ImageHDU      4        39    (2304, 1056)  float32
            .header    Header
            .data      ndarray
[4]      ('SCI', 3)    ImageHDU      5        37    (2304, 1056)  float32
            .header    Header
            .data      ndarray
[5]      ('VAR', 3)    ImageHDU      6        39    (2304, 1056)  float32
            .header    Header
            .data      ndarray
```

```
[6]     ('SCI', 4)    ImageHDU      7        39    (2304, 1056)  float32
           .header    Header
           .data      ndarray
```

The new extension pushed ('SCI',4) down and took its place in the sequence.

2.2.4 Removing an extension

Compared to appending and inserting extensions, removing them is a breeze. As before, the extension to remove can be speficied with the position number or with the extension name and version. Just remember that the position numbers are zero-indexed.

```
ad.remove(4)
ad.info()

ad.remove(('VAR',5))
ad.info()
```

After the two removal above, `ad` looks like this:

```
Filename: N20110313S0188.fits
    Type: AstroData
    Mode: update

AD No.    Name          Type       MEF No.  Cards    Dimensions   Format
         hdulist       HDUList
         phu           PrimaryHDU    0       179
         phu.header    Header
[0]      ('VAR', 1)    ImageHDU      1        39    (2304, 1056)  float32
           .header     Header
           .data       ndarray
[1]      ('SCI', 1)    ImageHDU      2        37    (2304, 1056)  float32
           .header     Header
           .data       ndarray
[2]      ('SCI', 2)    ImageHDU      3        37    (2304, 1056)  float32
           .header     Header
           .data       ndarray
[3]      ('VAR', 3)    ImageHDU      4        39    (2304, 1056)  float32
           .header     Header
           .data       ndarray
[4]      ('SCI', 4)    ImageHDU      5        39    (2304, 1056)  float32
           .header     Header
           .data       ndarray
```

2.2.5 Updating the existing file on disk

If a file has been opened in `update` mode, the file on disk can be overwritten with the `write()` command.

```
ad.filename
ad.write()
```

The first line will print the file name currently associated with the AstroData object, `ad`. This the file that will be written to.

More often though, the idea is to the write the modified output to a new file. This can be done regardless of the `mode` used when the file was opened. All that is needed is to specify a new file name. Note that this will change the file name associated with the AstroData object, permanently, any other `write` commands will write to the new file name.

```
ad.filename
ad.write('newfile.fits')
ad.filename
```

Before the write, the file name is `N20110313S0188.fits`. After the write, the file name is `newfile.fits`.

2.2.6 Closing and cleaning up

It is recommended to properly close the opened AstroData objects when they are no longer needed:

```
ad.close()
adread.close()
```

If you have been following along, the input file on disk was modified by one of the `write` examples above. We will need the unmodified file in the next section. To restore the file to the original:

```
import shutil
shutil.copy('../data_for_ad_user_manual/N20110313S0188.fits', '.')
```

2.3 Create New MEF Files

A new MEF file can be created from a copy of an existing file or created from scratch with AstroData objects.

2.3.1 Create New Copy of MEF Files

Let us consider the case where you already have a MEF file on disk and you want to work on it and write the modified MEF to a new file.

Basic example

Open a file, make modifications, write a new MEF file on disk:

```
1 from astrodata import AstroData
2
3 ad = AstroData('N20110313S0188.fits')
4 ... modifications here ...
5 ad.write('newfile2.fits')
6 ad.close()
```

Needing true copies in memory

Since in Python, and when working with AstroData objects, the memory can be shared between variables, it is sometimes necessary to create a "true" copy of an AstroData object to keep us from modifying the original.

By using `deepcopy` on an AstroData object the copy is a true copy, it gets given its own memory allocation. This allows one to modify the copy while leaving the original AstroData intact. This feature is useful when an operation requires both the modified and the original AstroData object since by design a simple copy or assignment points to a common location in memory. Use carefully however, your memory usage can grow rapidly if you over-use.

```
from astrodata import AstroData
from copy import deepcopy

ad = AstroData('N20110313S0188.fits')
adcopy = deepcopy(ad)
```

In the example above, `adcopy` is now completely independent copy of `ad`. This also means that you have doubled the memory footprint. Also note that both copies have the same file name associated to them; be mindful of that if you `write` the files back to disk.

2.3.2 Create New MEF Files from Scratch

Another use case is creating a new MEF files or AstroData object when none existed before. The pixel data needs to be created as a Numpy `ndarray`. The header must be created as PyFITS header. IMPORTANT: AstroData currently is not compatible with `astropy.io.fits`, one *must* use the standalone PyFITS module (it comes with Ureka).

```
1  from astrodata import AstroData
2  import pyfits as pf
3  import numpy as np
4
5  # Create an empty header.
6  new_header = pf.Header()
7
8  # Create a pixel data array.
9  new_data = np.linspace(0., 1000., 2048*1024).reshape(2048,1024)
10
11 # Create an AstroData object and give it a filename
12 new_ad = AstroData(data=new_data, header=new_header)
13 new_ad.filename = 'gradient.fits'
14
15 # Write the file to disk and close
16 new_ad.write()
17 new_ad.close()
```

The input header does not need to have anything in it (Line 6). In fact, if you are really creating from scratch, it is probably better to leave it empty and populate it after the creation of the AstroData object. Upon creation, AstroData, through PyFITS, will take care of adding the minimal set of header cards to make the file FITS compliant.

The pixel data array must be a `ndarray`. On Line 9, we create a 1024 x 2048 array, filled with a gradient of pixel value.

It is important to attach a name to the AstroData object (line 13) if it is to be written to disk. No default names are assigned to new AstroData objects.

CHAPTER

THREE

ASTRODATATYPES

3.1 What are AstroDataTypes

AstroDataTypes are AstroData's way to know about itself. When a file is opened with AstroData, the headers are inspected, data identification rules are applied, and all applicable AstroDataTypes are assigned. From that point on, the AstroData object "knows" whether it is a GMOS image, a NIRI spectrum, an IFU from GMOS or NIFS. This embedded knowledge is critical to the header keyword mapping done by the Descriptors (see *Section 4 - FITS Headers*), for examples. The RecipeSystem also depends heavily on the AstroDataType feature.

Examples of AstroDataTypes are: GMOS_IMAGE, SIDEREAL, GMOS_IFU_FLAT, NIRI_CAL, NIRI_SPECT, GEMINI_SOUTH, etc.

The AstroDataTypes can also refer to data processing status, eg. RAW. This feature is not used as much yet.

The types are obviously observatory and instrument dependent. The identification rules do need to be coded for AstroData to assign AstroDataTypes. This has been done for most if not all Gemini data. The Gemini Types are included in the `astrodata_Gemini` package that is installed along with `astrodata` when `gemini_python` is installed. Keeping the instrument rules and configuration separate from `astrodata` keeps it generic, and allows other packages to be easily added, for example, one might want to add a third-party package for CFHT instruments.

3.2 Using AstroDataTypes

Try it yourself

If you wish to follow along and try the commands yourself, download the data package, go to the `playground` directory and copy over the necessary files.

```
cd <path>/gemini_python_datapkg-X1/playground
cp ../data_for_ad_user_manual/N20111124S0203.fits .
```

Then launch the Python shell:

```
python
```

The attribute `types` is a common way to check the AstroDataType and make logic decisions based to on the type.

```
from astrodata import AstroData

ad = AstroData('N20111124S0203.fits')

if 'GMOS_IMAGE' in ad.types:
   print "I am a GMOS Image."
else:
   print "I am these types instead: ", ad.types
```

The attribute `types` is a list of all the AstroDataTypes associated with the dataset. Other than for controlling the flow of the program, it can be useful when interactively exploring the various types associated with a dataset, or when there's a need to write all the types to the screen or to a file, for logging purposes, for example.

If you have a set of Gemini datasets on disk and you wish to know which AstroDataTypes are associated with them, use the shell tool `typewalk` in that directory and you will be served a complete list of types for each datasets in that directory. Try it in the `data_for_ad_user_manual` directory. Open another shell (not the interactive Python shell) and

```
cd <path>/gemini_python_datapkg-X1/data_for_ad_user_manual
typewalk
```

You will get:

```
directory: /data/giraf/gemini_python_datapkg-X1/data_for_ad_user_manual
     estgsS20080220S0078.fits ......... (GEMINI) (GEMINI_SOUTH) (GMOS)
     ................................... (GMOS_LS) (GMOS_S) (GMOS_SPECT) (LS)
     ................................... (PREPARED) (SIDEREAL) (SPECT)
     gmosifu_cube.fits ................. (GEMINI) (GEMINI_SOUTH) (GMOS)
     ................................... (GMOS_IFU) (GMOS_IFU_BLUE)
     ................................... (GMOS_IFU_RED) (GMOS_IFU_TWO)
     ................................... (GMOS_S) (GMOS_SPECT) (IFU)
     ................................... (PREPARED) (SIDEREAL) (SPECT)
     N20110313S0188.fits ............... (GEMINI) (GEMINI_NORTH) (GMOS)
     ................................... (GMOS_IMAGE) (GMOS_N) (GMOS_RAW)
     ................................... (IMAGE) (RAW) (SIDEREAL) (UNPREPARED)
     N20110316S0321.fits ............... (CAL) (GEMINI) (GEMINI_NORTH) (GMOS)
     ................................... (GMOS_CAL) (GMOS_IMAGE)
     ................................... (GMOS_IMAGE_FLAT)
     ................................... (GMOS_IMAGE_TWILIGHT) (GMOS_N)
     ................................... (GMOS_RAW) (IMAGE) (RAW) (SIDEREAL)
     ................................... (UNPREPARED)
     N20111124S0203.fits ............... (CAL) (GEMINI) (GEMINI_NORTH) (GMOS)
     ................................... (GMOS_CAL) (GMOS_IFU) (GMOS_IFU_FLAT)
     ................................... (GMOS_IFU_RED) (GMOS_N) (GMOS_RAW)
     ................................... (GMOS_SPECT) (IFU) (RAW) (SIDEREAL)
     ................................... (SPECT) (UNPREPARED)
```

The attribute `types` returns all the processing status flags as well as the types proper. Those two concepts can be separated. The method `type()` returns only types, no status; the method `status()` returns only processing status, no types.

```
ad.type()
ad.status()
```

The `type()` statement returns:

```
['GMOS_IFU', 'GMOS_IFU_RED', 'IFU', 'GEMINI_NORTH', 'GMOS_N',
'GMOS_IFU_FLAT', 'GMOS_CAL', 'GEMINI', 'SIDEREAL', 'GMOS_SPECT',
'CAL', 'GMOS', 'SPECT']
```

and the `status()` statement returns:

```
['GMOS_RAW', 'UNPREPARED', 'RAW']
```

If code applies modifications to the AstroData object that result in changes to the AstroDataTypes, it is necessary to let the system know about it. The method `refresh_types()` rescan the AstroData headers and reapply the identification rules. This type refreshing is used mostly when the processing status needs to be changed, for example once the raw data has been standardized, it's processing status becomes "PREPARED".:

```
ad.refresh_types()
```

3.3 Creating New AstroDataTypes [Advanced Topic]

Todo

Primer on creating new AstroDataTypes.

Note: refer to programmer's manual, but give some idea of what needs to be done and the basic principles

CHAPTER

FOUR

FITS HEADERS

Try it yourself

If you wish to follow along and try the commands yourself, download the data package, go to the `playground` directory and copy over the necessary file.

```
cd <path>/gemini_python_datapkg-X1/playground
cp ../data_for_ad_user_manual/N20111124S0203.fits .
```

Then launch the Python shell:

```
python
```

4.1 AstroData Descriptors

AstroData Descriptors provide a "header keyword-to-concept" mapping that allows one to access header information in a consistent manner, regardless of which instrument the dataset is from. Like for the AstroDataTypes, the mapping is coded in a configuration package that is provided by the observatory or the user.

For example, if one were interested to know the filter used for an observation, normally one would need to know which specific keyword or set of keywords to look at for that instrument. However, once the concept of "filter" is coded in a Descriptor, one now only needs to call the `filtername` Descriptor to retrieve the information.

The Descriptors are closely associated with AstroDataTypes. The AstroDataType of the AstroData object will tell the Descriptor system which piece of the configuration package code to call to retrieve the information requested.

Note: If the Descriptors have not been configured for a dataset's specific AstroDataType, or if the AstroDataType for the data had not been defined, in the configuration package, the Descriptors will not work. In that case, it is nevertheless possible to access AstroData header information directly with the pyfits interface. This is also shown later in this chapter.

To get the list of descriptors available for an AstroData object:

```
1 from astrodata import AstroData
2
3 ad = AstroData('N20111124S0203.fits')
4 ad.descriptors
5 sorted(ad.descriptors.keys())
```

The `descriptors` property returns a dictionary with the name of the descriptors available for this AstroData object as the keys. The really interesting information is in the keys. On Line 5, we retrieve the sorted list of available Descriptors.

Most Descriptor names are readily understood, but one can get a short description of what the Descriptor refers to by calling the Python help function, for example:

```
help(ad.airmass)
```

Descriptors associated with standard FITS keywords are defined in the `ADCONFIG_FITS` package found in `astrodata_FITS`. All the Descriptors associated with other concepts used by the Gemini software are found in the `ADCONFIG_Gemini` package, part of `astrodata_Gemini`.

A user reducing Gemini data or coding for existing Gemini data only need to make sure that `astrodata_FITS` and `astrodata_Gemini` have been installed. A user coding for a new Gemini instrument, or for another observatory, will need to write the configuration code for the new Descriptors and AstrodataTypes. That is an advanced topic not covered by this manual.

4.2 Accessing Headers

4.2.1 Accessing headers with Descriptors

Whenever possible the Descriptors should be used to get information from the headers. This allows for maximum re-use of the code as it will work on any datasets with an AstroDataTypes. Here are a few examples using Descriptors:

```
1  from astrodata import AstroData
2  import numpy as np
3
4  ad = AstroData('N20111124S0203.fits')
5
6  #--- print a value
7  print 'The airmass is : ',ad.airmass()
8
9  #--- use a value to control the flow
10 if ad.exposure_time() < 240.:
11    print 'This is a short exposure'
12 else:
13    print 'This is a long exposure'
14
15 #--- multiply all extensions by their respective gain
16 print 'The average before: ', np.average(ad['SCI',2].data)
17 print 'The gain for [SCI,2]', ad['SCI',2].gain()
18
19 ad.mult(ad.gain())
20
21 print 'The average after: ', np.average(ad['SCI',2].data)
22
23 #--- do arithmetics
24 fwhm_pixel = 3.5
25 fwhm_arcsec = fwhm_pixel * ad.pixel_scale()
26
27 print 'The FWHM in arcsec is: ', fwhm_arcsec
```

The Descriptors are returned as DescriptorValue objects. In many types of statements the DescriptorValue will be automatically converted to the appropriate Python type based on context. All the Descriptor calls above do that convertion. For example, on Line 25, the DescriptorValue returned by `pixel_scale()` is converted to a Python float for the multiplication.

The AstroData arithmetics used on Line 19 will be discussed in more details in a later chapter. Essentially, the `gain()` Descriptor returns a DescriptorValue with gain information for each of the 6 extensions. Then AstroData's

`mult` method multiplies each pixel data extensions with the appropriate gain value for that extension. No looping necessary, AstroData and Descriptors are taking care of it.

When the automatic convertion to a Python cannot be determine from context the programmer must use the method `as_pytype()`.

```
ad.pixel_scale()
ad.pixel_scale().as_pytype()
```

The first line returns a DescriptorValue, the second line returns a float.

4.2.2 Accessing headers directly

Not all the header content has been mapped with Descriptors, nor should it. The header content is nevertheless accessible. With direct access, there are no DescriptorValue involved and the type returned matches what is stored in the header.

One important thing to keep in mind is that the PHU and the extension headers are accessed differently. The method `phu_get_key_value` accesses the PHU header; the method `get_key_value` accesses the header of the specified extension.

Here are some direct access examples:

```
from astrodata import AstroData

ad = AstroData('N20111124S0203.fits')

# Get keyword value from the PHU
aofold_position = ad.phu_get_key_value('AOFOLD')

# Get keyword value from a specific extension
print ad['SCI',1].get_key_value('NAXIS2')

# Get keyword value for all SCI extensions
for extension in ad['SCI']:
   naxis2 = extension.get_key_value('NAXIS2')
   print naxis2
```

4.2.3 Whole headers

Entire headers can be retrieve as PyFITS Header object.

```
# Get the header for the PHU
phuhdr = ad.phu.header

# Get the header for extension SCI, 1
exthdr = ad['SCI',1].header
```

In the interactive Python shell, listing the header contents to screen can be done as follow:

```
# For the PHU
ad.phu.header

# For a specific extension:
ad['SCI',2].header

# For all the extensions:  (PHU excluded)
ad.headers
```

4.2.4 EXTNAME and EXTVER

MEF files have the concept of naming and versioning extensions. The header keywords storing the name and version are `EXTNAME` and `EXTVER`. AstroData uses that concept extensively. In fact, even if a MEF on disk does not have `EXTNAME` and `EXTVER` defined, for example Gemini raw datasets, upon opening the file AstroData will assign names and versions to each extension. The default behavior is to assign all extension a `EXTNAME` of `SCI` and then version them sequential from 1 to the number of extension present.

The name and version of an extension is obtained this way:

```
name = ad[1].extname()
version = ad[1].extver()
print name, version
```

4.3 Updating and Adding Headers

Header cards can be updated or added to the headers. As for the simple access to the headers, there are methods to work on the PHU and different methods to work on the extensions.

The methods to update and add headers mirror the access methods. The method `phu_set_key_value()` modifies the PHU. The method `set_key_value()` modifies the extension headers.

The inputs to the `phu_set_key_value` and `set_key_value` methods are *keyword*, *value*, *comment*. The comment is optional.

```
from astrodata import AstroData

ad = AstroData('N20111124S0203.fits')

# Add a header card to the PHU
ad.phu_set_key_value('MYTEST', 99, 'Some meaningless keyword')

# Modify a header card in the second extension
ad[1].set_key_value('GAIN',5.)

# The extension can also be specified by name and version.
ad['SCI',2].set_key_value('GAIN', 10.)

# In a loop
for extension in ad['SCI']:
   extension.set_key_value('TEST',9, 'This is a test.')
```

The `set_key_value` method works only on an extension, it will not work on the whole AstroData object. For example, the following **will not** work:

```
# DOES NOT WORK.  An extension must be specified.
ad.set_key_value('TEST', 8, 'This test does not work')
```

4.3.1 EXTNAME and EXTVER

The name and version of an extension are so critical to AstroData that, like for access, the editing of `EXTNAME` and `EXTVER` is done through a special method.

The name and version of an extension can be set or reset manually with the `rename_ext` method:

```
ad['SCI',1].rename_ext('VAR',4)
```

Be careful with this function. Having two extensions with the same name and version in an AstroData data object, or a MEF files for that matter, can lead to strange problems.

4.4 Adding Descriptors for New Instruments [Advanced Topic]

Todo

Primer on Descriptor definitions for new instrument.

Note: refer to Descriptors document for complete instructions

CHAPTER

FIVE

PIXEL DATA

Try it yourself

If you wish to follow along and try the commands yourself, download the data package, go to the `playground` directory and copy over the necessary files.

```
cd <path>/gemini_python_datapkg-X1/playground
cp ../data_for_ad_user_manual/N20110313S0188.fits .
cp ../data_for_ad_user_manual/gmosifu_cube.fits .
cp ../data_for_ad_user_manual/estgsS20080220S0078.fits .
```

Then launch the Python shell:

```
python
```

5.1 Operate on the Pixel Data

The pixel data are stored in a numpy `ndarray`. Therefore anything that can be done with numpy on a `ndarray` can be done on the pixel data stored in the AstroData object. Examples include arithmetic, statistics, display, plotting, etc. Please refer to numpy documentation for details on what it offers. In this chapter, we will present some typical examples.

But first, here is how to acess the data array stored in an AstroData object.

```
1  from astrodata import AstroData
2
3  ad = AstroData('N20110313S0188.fits')
4
5  the_data = ad['SCI',2].data
6
7  # Loop through the extensions.
8  for extension in ad['SCI']:
9     the_data = extension.data
10    print the_data.sum()
```

An extension's `data` attribute returns a numpy `ndarray`. An extension must be specified by name (Line 5) or position ID, or extracted from the main AstroData object (Line 8). Also, remember that PHUs do not have any pixel data, so none of what is discussed in this chapter applies to the PHU.

The AstroData pixel data can be manipulated like any other `ndarray`. For example, on Line 10, we calculate the sum of the pixel values with the `ndarray sum()` method.

5.2 Arithmetic on AstroData Objects

AstroData supports basic arithmetic directly:

addition	.add()
subtraction	.sub()
multiplication	.mult()
division	.div()

The big advantage of using the AstroData implementation of those operator is that if the AstroData object has variance and data quality planes, those will be calculated and propagated to the output appropriately.

```
from astrodata import AstroData

ad = AstroData('N20110313S0188.fits')

# addition
#   ad = ad + 5.
ad.add(5.)

# subtraction
#   ad = ad - 5.
ad.sub(5.)

# multiplication.  Using descriptor as operand.
#   ad = ad * gain
ad.mult(ad.gain())

# division. Using descriptor as operand.
#   ad = ad / gain
ad.div(ad.gain())
```

When using AstroData arithmetic, all the science frames (`EXTNAME='SCI'`) are operated on. The modifications are **done in-place**, the AstroData object is modified.

The AstroData arithmetic methods can be stringed together. Note that because the calculations are done "in-place", **operator precedence cannot be respected**. For example,

```
ad.add(5).mult(10).sub(5)

# means:  ad = ((ad + 5) * 10) - 5
# not: ad = ad + (5 * 10) - 5
```

The AstroData data arithmetic method modify the data "in-place". This means that the data values are modified and the original values are no more. If you need to keep the original values unmodified, for example, you will need them later, use `deepcopy` to make a separate copy on which you can work.

Let us say that we want to calculate `y = x*10 + x` where `x` is the pixel values. We must use `deepcopy` here since after `ad.mult(10)` the values in `ad` will have been modified and cannot be used for the `+ x` part of the equation.

Let us follow a pixel through the math.

The WRONG way to calculate `x*10 + x`:

```
from astrodata import AstroData

ad = AstroData('N20110313S0188.fits')

value_before = ad['SCI',1].data[50,50]
expected_value_after = value_before*10 + value_before
```

```

ad.mult(10).add(ad)
bad_value_after = ad['SCI',1].data[50,50]

print expected_value_after, bad_value_after

ad.close()

# The result of the arithmetic above is y = (x*10) + (x*10)
```

The CORRECT way to calculate `x*10 + x`:

```
from astrodata import AstroData
from copy import deepcopy

ad = AstroData('N20110313S0188.fits')
adcopy = deepcopy(ad)

value_before = ad['SCI',1].data[50,50]
expected_value_after = value_before*10 + value_before

ad.add(adcopy.mult(10))

good_value_after = ad['SCI',1].data[50,50]
print expected_value_after, good_value_after

ad.close()
adcopy.close()
```

As one can see, for complex equation, using the AstroData arithmetic method can get fairly confusing. Operator overload would solve this situation but it has not been implemented yet mostly due to a lack of resources. Therefore, we recommend the use numpy for really complex equation since operator overload is implemented in numpy and the operator precedence is respected. The downside is that if you need the variance plane propagated correctly, you will have to do the math yourself.

Here is the `y = x*10 + x` operation again, but this time numpy is used on the `ndarray` returned by `.data`. Like before, we follow a pixel through the math.

```
from astrodata import AstroData

ad = AstroData('N20110313S0188.fits')

value_before = ad['SCI',1].data[50,50]
expected_value_after = value_before*10 + value_before

for extension in ad['SCI']:
    data_array = extension.data
    data_array = data_array*10 + data_array
    extension.data = data_array

value_after = ad['SCI',1].data[50,50]
print expected_value_after, value_after

ad.close()
```

5.3 Variance

The AstroData arithmetic methods can propagate the variance planes, if any are present. The variance extensions must be named `VAR` to be recognized as such.

The initial variance from read noise and poisson noise normally needs to be calculated by the programmer; the raw data normally contains only science extensions.

5.3.1 Adding variance extensions

For the sake of simplicity, only the poisson noise is considered in this example.

```
1  from astrodata import AstroData
2
3  ad = AstroData('N20110313S0188.fits')
4  ad.info()
5
6  for extension in ad['SCI']:
7     variance = extension.data / extension.gain().as_pytype()
8     variance_header = extension.header
9     variance_extension = AstroData(data=variance, header=variance_header)
10    variance_extension.rename_ext('VAR')
11    ad.append(variance_extension)
12
13 ad.info()
14
15 # Let's save a copy of this dataset.
16 ad.write('N188_with_var.fits')
17 ad.close()
```

On Line 6, the loop through all the science extension is launched. The Poisson noise will be calculated for each science extension and stored in a new extension named 'VAR'. The extension version informs on the association between the 'SCI' and the 'VAR' extensions, eg. ['VAR', 1] is the variance for ['SCI', 1].

For each science extension, the variance is calculated from the pixel data and the gain obtained from the Descriptor `.gain` (Line 7). Note the use of `as_pytype()` on the Descriptor. Since `extension.data` is a `ndarray` not a standard Python type, the DescriptorValue does not know how it is expected to behave, requiring the use of `as_pytype()` which converts the DescriptorValue to a Python float.

On Line 8, we simply copy the header for the science extension and use that as the header for the new variance extension (Line 9).

The new variance extension is renamed to 'VAR' on Line 10 – it was 'SCI' since we copied the header – and append the extension to the AstroData object (Line 11).

Finally, we write that AstroData object to a new MEF on disk. We will use that MEF in the next examples.

For reference, the AstroData object before the variance planes are added looks like this:

```
Filename: N20110313S0188.fits
    Type: AstroData
    Mode: readonly

AD No.    Name          Type       MEF No.  Cards    Dimensions   Format
         hdulist       HDUList
         phu           PrimaryHDU    0       179
         phu.header    Header
[0]      ('SCI', 1)    ImageHDU      1        37    (2304, 1056)  float32
```

```
            .header    Header
            .data      ndarray
[1]      ('SCI', 2)    ImageHDU      2        37    (2304, 1056)  float32
            .header    Header
            .data      ndarray
[2]      ('SCI', 3)    ImageHDU      3        37    (2304, 1056)  float32
            .header    Header
            .data      ndarray
```

After the variance planes are added, the structure looks like this:

```
Filename: N20110313S0188.fits
    Type: AstroData
    Mode: readonly

AD No.    Name          Type      MEF No.  Cards    Dimensions   Format
         hdulist       HDUList
         phu           PrimaryHDU    0       179
         phu.header    Header
[0]      ('SCI', 1)    ImageHDU      1        37    (2304, 1056)  float32
            .header    Header
            .data      ndarray
[1]      ('SCI', 2)    ImageHDU      2        37    (2304, 1056)  float32
            .header    Header
            .data      ndarray
[2]      ('SCI', 3)    ImageHDU      3        37    (2304, 1056)  float32
            .header    Header
            .data      ndarray
[3]      ('VAR', 1)    ImageHDU      4        37    (2304, 1056)  float32
            .header    Header
            .data      ndarray
[4]      ('VAR', 2)    ImageHDU      5        37    (2304, 1056)  float32
            .header    Header
            .data      ndarray
[5]      ('VAR', 3)    ImageHDU      6        37    (2304, 1056)  float32
            .header    Header
            .data      ndarray
```

5.3.2 Automatic variance propagation

As mentioned before, if the AstroData arithmetic methods are used, the variance will be propagated automatically. A simple `ad.mult()` suffices to multiply the science pixels and calculate the resulting variance, for all extensions.

Let us follow a science pixel and a variance pixel through the AstroData arithmetic.

```
#     output = x * x
# var_output = var * x^2 + var * x^2

from astrodata import Astrodata

ad = AstroData('N188_with_var.fits')

value_before = ad['SCI',1].data[50,50]
variance_before = ad['VAR',1].data[50,50]
expected_value_after = value_before + value_before
expected_variance_after = 2 * (variance_before * value_before * value_before)

```

```
13 ad.mult(ad)
14
15 value_after = ad['SCI',1].data[50,50]
16 variance_after = ad['VAR',1].data[50,50]
17 print expected_value_after, value_after
18 print expected_variance_after, variance_after
19
20 ad.close()
```

So all it took to multiply the science extensions by themselves and propagate the variance accordingly was `ad.mult(ad)` (Line 13).

5.3.3 Manual propagation with numpy

To do the same thing as `ad.mult(ad)`, but by operating directly on the numpy arrays of each extension:

```
1  from astrodata import AstroData
2
3  ad = AstroData('N188_with_var.fits')
4
5  # This loop is the equivalent of ``ad.mult(ad)``
6  for i in range(1,ad.count_exts('SCI')+1):
7      d = ad['SCI',i].data
8      v = ad['VAR',i].data
9      data = d*d
10     variance = v * d*d + v * d*d
11     ad['SCI',i].data = data
12     ad['VAR',i].data = variance
13
14 ad.close()
```

5.4 Display

Displaying `ndarray` arrays from Python is straighforward with the `numdisplay` module. The module also has a function to read the position the cursor, which can be useful when developing an interactive tool.

The `numdisplay` module is a module of the `stsci.tools` package distributed in Ureka.

5.4.1 Displaying

To display the pixel data of an AstroData extension, the `numdisplay.display` function is used. A display tool, like DS9 or ximtool, must also be running.

```
1  from astrodata import AstroData
2  from stsci.numdisplay import display
3
4  ad = AstroData('N20110313S0188.fits')
5
6  display(ad['SCI',1].data)
7
8  # To scale "a la IRAF"
9  display(ad['SCI',1].data, zscale=True)
10
```

```
11 # To set the minimum and maximum values to display
12 display(ad['SCI',1].data, z1=700, z2=2000)
```

`numdisplay.display` accepts various arguments. See `help(display)` to get more information. The examples on Line 6, 9, and 12, are probably the most common, especially for users coming from IRAF.

5.4.2 Retrieving cursor position

The function `numdisplay.readcursor` can be used to retrieve cursor position. Note that it will **not** respond to mouse clicks, **only** keyboard entries are acknowledged.

When invoked, `readcursor()` will stop the flow of the program and wait for the user to put the cursor on top of the image and type a key. A **string** with four space-separated values are going to be returned: the x and y coordinates, a frame reference number, and the value of the key the user hit.

```
1  # here we assume that the previous example has just been run.
2
3  from stsci.numdisplay import readcursor
4
5  # User instructions: Put cursor on image, type a key.
6  cursor_coo = readcursor()
7  print cursor_coo
8
9  # To extract only the x,y coordinates:
10 (xcoo, ycoo) = cursor_coo.split()[:2]
11 print xcoo, ycoo
12
13 # If you are also interested in the keystoke:
14 (xcoo, ycoo, junk, keystroke) = cursor_coo.split()
15 print 'You pressed this key: "%s"' % keystroke
```

5.5 Useful tools from the Numpy and SciPy Modules

Like the Display section, this section is not really specific to AstroData, but is rather an introduction to numpy and scipy, and to using those modules on `ndarray` objects. Since AstroData pixel data is stored in that format, it is believe important to show a few examples to steer new users in the right direction.

The `numpy` and `scipy` modules offer a multitude of functions and tools. They both have their own documentation. Here we simply highlighting a few functions that could be used for common things an astronomer might want to do. The idea is to get the reader started in her exploration of `numpy` and `scipy`.

5.5.1 ndarray

```
1  from astrodata import AstroData
2  import numpy as np
3
4  ad = AstroData('N20110313S0188.fits')
5  data = ad['SCI',2].data
6
7  # Shape of array, (NAXIS2, NAXIS1)
8  data.shape
9
10 # Value of pixel with IRAF coordinates (100, 50)
```

```
11 data[49,99]
12
13 # Data type
14 data.dtype
```

The two most important thing to remember for users coming from the IRAF world are that the array has the y-axis in the first index, the x-axis in the second (Line 8, 11), and that the array indices are zero-indexed, not one-indexed (Line 11).

Sometimes it is useful to know the type of the values stored in the array, eg. integer, float, double-precision, etc., this information is obtained with `dtype` (Line 14).

5.5.2 Simple numpy statistics

A lot of functions and methods are available in numpy to probe the array, too many to cover here, but here are a couple examples.

```
data.mean()
np.average(data)
np.median(data)
```

Note how `mean()` is called differently from the other two. `mean()` is a `ndarray` method, the others are numpy functions. The implementation details are clearly well beyond the scope of this manual, but when looking for the tool you need, keep in mind that there are two sets of functions to look into. Duplications like `.mean()` and `np.average()` can happen, but they are not the norm. The readers are strongly encouraged to refer to the numpy documentation to find the tool they need.

5.5.3 Clipped statistics

It is common in astronomy to apply clipping to the statistics, a clipped average, for example.

The numpy `ma` module can be used to create masks of the values to reject.

In the example below, we calculate a clipped mean with rejection at +/- 3 times the standard deviation.

```
1 import numpy.ma as ma
2
3 stddev = data.std()
4 mean = data.mean()
5
6 clipped_mean = ma.masked_outside(data, mean-3*stddev, mean+3*stddev).mean()
```

On Line 6, `ma.masked_outside` identify all values falling outside the allowed range, and creates a `MaskedArray` which is a combination of the data array and a True/False mask, with True identifying the values to reject. The really convenient thing is that the method `mean` will take this information and calculate the average ignoring all values tagged for rejection.

WARNING: The `numpy.clip` function is not the equivalent of the mask solution. As explain in the `numpy.clip` documentation, the function *replaces* the extreme values with the minimum and maximum, it *does not reject* the extreme values.

5.5.4 Filters with scipy

Another common operation is the filtering of an image, for example convolving with a gaussian filter. The scipy module `ndimage.filters` offers several such functions for image processing. See the scipy documentation for

more details.

The example below applies a gaussian filter to a pixel array.

```
1  import scipy.ndimage.filters as filters
2  from stsci.numdisplay import display
3
4  convolved_data = np.zeros(data.size).reshape(data.shape)
5  sigma = 10.
6  filters.gaussian_filter(data, sigma, output=convolved_data)
7
8  # visually compare the convolve image with the original.
9  display(data, zscale=True)
10 display(convolved_data, zscale=True, frame=2)
11
12 # To put the convolved data back in the AstroData object
13 ad['SCI',2].data = convolved_data
```

On Line 4, a numpy array of the same size and shape as the input data is created and filled with zeros. This receives the output convolved image produced by the function `gaussian_filter` (Line 6).

On Line 13, the input data is replaced with the convolved data. Remember that one will need to use `write` to make that change effective on disk.

5.5.5 Many other tools

The world of `numpy`, `scipy`, and the new `astropy` is rich and vast. The reader should refer to those packages' documentation to learn more.

5.6 Using the AstroData Data Quality Plane

Todo

Write examples that use the DQ plane. Eg. transform DQ plane in a numpy mask and do statistics.

5.7 Manipulate Data Sections

Sections of the data array can be accessed and processed. It is important to note here that when indexing a numpy array, the left most number refers to the highest dimension's axis (eg. in IRAF sections are in (x,y) format, in Python they are in (y,x) format). Also important is to remember that the numpy arrays are 0-indexed, not 1-indexed like in Fortran or IRAF. For example, in a 2-D numpy array, the pixel position (x,y) = (50,75) would be accessed as data[74,49].

5.7.1 Basic statistics on section

```
1  from astrodata import AstroData
2  import numpy as np
3
4  ad = AstroData('N20110313S0188.fits')
5  data = ad['SCI',2].data
6
7  # Get statistics for a 25x25 pixel-wide box centered on pixel 50,75.
```

```
8  mean = data[62:87,37:62].mean()
9  median = np.median(data[62:87,37:62])
10 stddev = data[62:87,37:62].std()
11 minimum = data[62:87,37:62].min()
12 maximum = data[62:87,37:62].max()
13
14 print "Mean    Median Stddev  Min    Max\n", \
15    mean, median, stddev, minimum, maximum
```

There is one odd thing that the reader should notice. On Line 9, `median` is not being called like the others. This is because there is no `median` method associated with a `ndarray`. But there is a numpy function, so that is what is used.

5.7.2 Example - Overscan subtraction

Several concepts from previous chapters are used in this example. The Descriptors are used to retrieve overscan section and data section information from the headers. Numpy statistics is done on the pixel data sections. AstroData arithmetics is used to subtract the overscan level. Finally, the overscan section is trimmed off and the AstroData is written to a new file.

To make the example more complete, we use the file created in the Variance section. We will propagate the variance and trim the variance planes too.

```
1  from astrodata import AstroData
2
3  ad = AstroData('N188_with_var.fits')
4
5  # Get EXTVER-keyed dictionary for the overscan section and the data
6  # section.
7  oversec_dict = ad.overscan_section().collapse_by_extver()
8  datasec_dict = ad.data_section().collapse_by_extver()
9
10 # Loop through the science extensions.
11 for ext in ad['SCI']:
12    #
13    extver = ext.extver()
14    #
15    (x1, x2, y1, y2) = oversec_dict[extver]
16    (dx1, dx2, dy1, dy2) = datasec_dict[extver]
17    #
18    # Measure the overscan level
19    mean_overscan = ext.data[y1:y2,x1:x2].mean()
20    #
21    # Append variance and subtract overscan. Variance is propagated.
22    ext.append(ad['VAR',extver])
23    ext.sub(mean_overscan)
24    #
25    # Trim the data to remove the overscan section and keep only
26    # the data section.
27    ext['SCI',extver].data = ext['SCI',extver].data[dy1:dy2,dx1:dx2]
28    ext['VAR',extver].data = ext['VAR',extver].data[dy1:dy2,dx1:dx2]
29
30 ad.write('N188_overscanTrimmed.fits')
```

The dataset loaded in Line 3 has three 'SCI' extensions and three 'VAR' extensions.

On Lines 7 and 8, the descriptors `overscan_section` and `data_section` are used to retrieve the region information relating to the location of the overscan section and data section in each extension. The return values for each

extension is a list of four zero-indexed indices identifying the x and y axes lower and upper range for the section.

Since the overscan and data sections are the same for a given extension version regardless of the extension name (ie. 'SCI' and 'VAR' have identical size and structure), the section information is keyed by extension version only with the help of the `collapse_by_extver()` method (Line 7 and 8).

The content of oversec_dict is

```
1: [0, 32, 0, 2304], 2: [0, 32, 0, 2304], 3: [1024, 1056, 0, 2304]}
```

The overscan section and the data section could have been obtained in the loop for each extension. It is however more efficient to retrieve the descriptor information once than three times in the loop, when it is possible to do so.

Moving on to the loop starting on Line 11. For convenience, the extension version of the current extension is stored in a variable on Line 13. It will be used to access the section dictionaries from Line 7 and 8, and to associate a variance extension to the current extension.

The boundaries of the overscan section and of the data section for the current extension are assigned to convenient variables on Lines 15 and 16. The values are zero-indexed to match the `ndarray`.

The overscan is here simply calculated as the average within the overscan section (Line 19).

To simplying our lifes, we want to be using the automatic variance propagation offered by the AstroData arithmetic utility. The extension obtained from the `in` on Line 11 is an AstroData object with only the 'SCI' extension. No variance propagation will occur if there are no 'VAR' extension in that AstroData object. Therefore, on Line 22, we append the 'VAR' extension of the same version as the 'SCI' extension. When the subtraction is done on Line 23, the operation affects both the 'SCI' extension and the newly appended 'VAR' extension.

Now that the overscan level has been subtracted, that section needs to be trimmed off. Only the data section is kept. This is done on Lines 27 and 28. Note that after the 'VAR' extension is appended, there are two extensions in the AstroData object, and in order to work with them, they must now be specified. Contrast that with the calls on Lines 13 and 19; then there were still only one extension, the 'SCI' extension, and selection was implicit.

A final but important reminder. All of the work was done on the extensions extracted from the original AstroData object, `ad`. Since the extensions are not true copies but rather references to the original source, all the modifications to `ext` are in reality modifications to `ad`. So, now, all we have to do is write the modified `ad` to disk with another name (Line 31).

5.8 Data Cubes

Reduced Integral Field Unit (IFU) data is commonly stored in a cube, a three-dimensional array. The `data` component of an AstroData object can be such a cube, and can be manipulated and explored with numpy, plotted with matplotlib (the next section shows a few more matplotlib examples.)

In the MEF file, the x and y axes are in the first and second dimension, and the third dimension is for the wavelength axis. In the `ndarray`, that order is reversed. The dimensions of the `ndarray` are (wavelength, y, x).

```
1  from astrodata import AstroData
2  import numpy as np
3  from stsci.numdisplay import display
4  import matplotlib.pyplot as plt
5
6  adcube = AstroData('gmosifu_cube.fits')
7  adcube.info()
8
9  # The shape of the cube, (wavelength, y, x).
10 adcube.data.shape
11
```

```
12 # Sum along the wavelength axis to create a "white light" image.
13 sum_image = adcube.data.sum(axis=0)
14 display(sum_image, zscale=True)
15
16 # Plot a 1-D spectrum from pixel position (7,30)
17 plt.plot(adcube.data[:,29,6])
18 plt.show()
19
20 # Plot the same thing but with wavelength along the x axis of the plot.
21 crval3 = adcube.get_key_value('CRVAL3')
22 cdelt3 = adcube.get_key_value('CDELT3')
23 spec_pixel_length = adcube.data[:,29,6].size
24 wavelength = crval3 + np.arange(spec_pixel_length)*cdelt3
25 plt.clf()
26 plt.plot(wavelength, adcube.data[:,29,6])
27 plt.show()
```

On Line 13, the cube is "collapsed" in the wavelength dimension to produce a "white light" 2-D image.

On Line 17, a vector through the cube at pixel position (7,30) is retrieved and plotted with matplotlib. The pixel values are plotted against the zero-indexed pixel position along the wavelength axis. The `show()` statement (Line 18) might or might not be needed depending on your `interactive` setting in the `matplotlibrc` file.

To plot the spectrum with physical units on the plot's x-axis one needs the WCS information from the header (Lines 21 to 24). The statement `plt.clf()` simply clears the figure.

People familiar with astropy might prefer to use the `wcs` module to convert pixels to wavelenths (Lines 21 to 24).

5.9 Plot Data

In Python, the main tool to create plots is `matplotlib`. We have used it in the previous section on data cubes. Here we do not aimed at covering all of `matplotlib`; the reader should refer to that package's documentation. Rather we will give a few examples that might be of use for quick inspection of the data.

```
1  from astrodata import AstroData
2  import matplotlib.pyplot as plt
3  from matplotlib.colors import LogNorm
4
5  adimg = AstroData('N20110313S0188.fits')
6  adspec = AstroData('estgsS20080220S0078.fits')
7
8  # Line plot from image.  Row #1044.
9  line_index = 1043
10 line = adimg['SCI',2].data[line_index, :]
11 plt.clf()
12 plt.plot(line)
13 plt.show()
14
15 # Column plot from image, averaging across 11 pixels around column #327.
16 col_index = 326
17 width = 5
18 col_section = adimg['SCI',2].data[:,col_index-width:col_index+width+1]
19 column = col_section.mean(axis=1)
20 plt.clf()
21 plt.plot(column)
22 plt.show()
23
```

```
# Contour plot for section
galaxy = adimg['SCI',2].data[1045:1085,695:735]
plt.clf()
plt.imshow(galaxy, cmap='binary', norm=LogNorm())
plt.contour(galaxy)
plt.axis('equal')
plt.show()

# Spectrum in pixel
plt.clf()
plt.plot(adspec['SCI',1].data)
plt.show()

# Spectrum in wavelength (CRPIX1 = 1)
crpix1 = adspec['SCI',1].get_key_value('CRPIX1')
crval1 = adspec['SCI',1].get_key_value('CRVAL1')
cdelt1 = adspec['SCI',1].get_key_value('CDELT1')
length = adspec['SCI',1].get_key_value('NAXIS1')
wavelengths = crval1 + (np.arange(length)-crpix1+1)*cdelt1
plt.clf()
plt.plot(wavelengths, adspec['SCI',1].data)
plt.show()
```

CHAPTER

SIX

TABLE DATA

AstroData does not provide any special wrappers for FITS Table Data. But since AstroData is built on top of PyFITS, the standard PyFITS table functions can be used. The reader should refer to the PyFITS table documentation for complete details (e.g. https://pythonhosted.org/pyfits/users_guide/users_table.html). In this chapter, a few useful examples of basic usage are shown. As the reader will see, table manipulation can be tedious and tricky.

`astropy.table` might simplify some of the examples presented in this chapter. This will be looked into at some future time. (Suggestions for improvements are welcome.)

Try it yourself

If you wish to follow along and try the commands yourself, download the data package, go to the `playground` directory and copy over the necessary files.

```
cd <path>/gemini_python_datapkg-X1/playground
cp ../data_for_ad_user_manual/estgsS20080220S0078.fits .
```

Then launch the Python shell:

```
python
```

6.1 Read from a FITS Table

A FITS table is stored in a MEF file as a `BinTableHDU`. The table data is retrieved from the `AstroData` object with the same `.data` attribute as for pixel extensions, but for FITS tables `.data` returns a `FITS_rec`, which is a PyFITS class, instead of a Numpy `ndarray`. Here is how to get information out of a FITS table.

```
from astrodata import AstroData

adspec = AstroData('estgsS20080220S0078.fits')
adspec.info()

# For easier reference, assign FITS table to variable.
tabledata = adspec['MDF'].data

# Get the column names with 'names' or more details with 'columns'
tabledata.names
tabledata.columns

# Get all the data for a column
x_ccd_values = tabledata.field('x_ccd')
third_col = tabledata.field(2)

# Print the table content
```

```
18 print tabledata
19
20 # Print the first 2 rows
21 print tabledata[:2]
22
23 # Select rows based on some criterion
24 select_tabledata = tabledata[tabledata.field('y_ccd') > 2000.]
25 print select_tabledata
```

The first extension in that file is a FITS table with `EXTNAME` MDF, and `EXTVER` 1 (Lines 4, 7) . MDF stands for "Mask Definition File". In Gemini data, those are used in the data reduction to identify, to first order, where spectra fall on the detector.

The output of `names` (Line 10) is a simple list of strings. The output of `columns` is a PyFITS `ColDefs` object. When printed it looks like this:

```
ColDefs(
    name = 'ID'; format = '1J'; null = -2147483647; disp = 'A3'
    name = 'x_ccd'; format = '1E'; disp = 'F8.2'
    name = 'y_ccd'; format = '1E'; disp = 'F8.2'
    name = 'slittype'; format = '20A'; disp = 'A10'
    name = 'slitid'; format = '1J'; null = -2147483647; disp = 'A3'
    name = 'slitpos_mx'; format = '1E'; disp = 'F8.2'
    name = 'slitpos_my'; format = '1E'; disp = 'F8.2'
    name = 'slitsize_mx'; format = '1E'; disp = 'F8.2'
    name = 'slitsize_my'; format = '1E'; disp = 'F8.2'
    name = 'slittilt_m'; format = '1E'; disp = 'F8.2'
    name = 'slitsize_mr'; format = '1E'; disp = 'F6.2'
    name = 'slitsize_mw'; format = '1E'; disp = 'F6.2'
)
```

When a column is retrieved, like in Lines 14 and 15, the returned value is a numpy `ndarray`.

Note on Line 15 that the third column is in index position 2; all Python arrays are zero-indexed.

6.2 Create a FITS Table

Creating a FITS table is mostly a matter of creating the columns, name and data. The name is a string, the data is stored in a numpy `ndarray`.

```
from astrodata import AstroData
import pyfits as pf
import numpy as np

# Create the input data
snr_id = np.array(['S001','S002','S003'])
feii = np.array([780.,78.,179.])
pabeta = np.array([740.,307.,220.])
ratio = pabeta/feii

# Create the columns
col1 = pf.Column(name='SNR_ID', format='4A', array=snr_id)
col2 = pf.Column(name='ratio', format='E', array=ratio)
col3 = pf.Column(name='feii', format='E', array=feii)
col4 = pf.Column(name='pabeta', format='E', array=pabeta)

# Assemble the columns
```

```
cols = pf.ColDefs([col1, col2, col3, col4])

# Create the table HDU
tablehdu = pf.new_table(cols)

# Create an AstroData object to contain the table
# and write to disk.
new_ad = AstroData(tablehdu)
new_ad.rename_ext('MYTABLE', 1)
new_ad.info()

new_ad.write('mytable.fits')
```

A new FITS table can also be appended to an already existing AstroData object with the `.append()` function.

6.3 Operate on a FITS Table

The PyFITS manual is the recommended source for more complete documentation on working on FITS table with Python. Here are a few examples of how one can modify a FITS table.

6.3.1 Preparation for the examples

In order to run the examples in the next few sections, the reader will need to create these three tables.

```
from astrodata import AstroData
import pyfits as pf
import numpy as np

# Let us first create tables to play with
snr_id = np.array(['S001','S002','S003'])
feii = np.array([780.,78.,179.])
pabeta = np.array([740.,307.,220.])
ratio = pabeta/feii
col1 = pf.Column(name='SNR_ID', format='4A', array=snr_id)
col2 = pf.Column(name='ratio', format='E', array=ratio)
col3 = pf.Column(name='feii', format='E', array=feii)
col4 = pf.Column(name='pabeta', format='E', array=pabeta)
cols_t1 = pf.ColDefs([col1,col3])
cols_t2 = pf.ColDefs([col1,col4])
cols_t3 = pf.ColDefs([col2])

table1 = pf.new_table(cols_t1)
table2 = pf.new_table(cols_t2)
table3 = pf.new_table(cols_t3)
```

6.3.2 Merging tables

WARNING: The input tables must **not** share any common field (ie. column) names. For example, *table1* and *table2* created above cannot be merged this way since they share `col1`.

The merging of tables is effectively the equivalent of appending columns.

```
1 merged_cols = table1.columns + table3.columns
2 merged_table = pf.new_table(merged_cols)
3
4 merged_table.columns.names  # or merged_table.data.names
5 print merged_table.data
```

The columns are now:

```
['SNR_ID', 'feii', 'ratio']
```

It is interesting to note that table operations are actually *column* operations followed by the creation of a new table (Lines 1 and 2). The next example will illustrate this a bit better.

6.3.3 Appending and deleting columns

```
1 # Append the 'pabeta' column from table2 to table1
2 index_of_pabeta_col = table2.columns.names.index('pabeta')
3 table1.columns.add_col(table2.columns[index_of_pabeta_col])
4 table1 = pf.new_table(table1.columns)
5
6 table1.columns.names
7 print table1.data
```

The append example (Lines 2-4) shows that the real work is done on the columns, not on the table as such. To add a column to `table1`, once the columns have been reorganized, a *new* table is created and, in this case, replaces the original `table1`.

The index of the `pabeta` column in `table2` is found with the `index` method as shown on Line 2. Then it is just a matter of adding that column from `table2` to the columns of `table1` (Line 3).

The columns in the new `table1` are:

```
['SNR_ID', 'feii', 'pabeta']
```

```
1 #   To "delete" the 'pabeta' column from this new table1
2 table1.columns.del_col('pabeta')
3 table1 = pf.new_table(table1.columns)
4
5 table1.columns.names
6 print table1.data
```

To delete a column, the process is similar: the work is done on the columns, then a *new* table is created to replace the original (Lines 2, 3).

The columns in the final `table1` are:

```
['SNR_ID', 'feii']
```

6.3.4 Inserting columns

Column insertion is really about gathering all the columns and reorganizing them manually. There are no "insertion" tool, per se, in pyfits. (`astropy.table` does have one though.)

Below, we insert the column from `table3` in-between the first and second column of `table1`.

```
t1_col1 = table1.columns[0]
t1_col2 = table1.columns[1]
t3_col1 = table3.columns[0]
table1 = pf.new_table([t1_col1,t3_col1,t1_col2])

table1.columns.names
print table1.data
```

The columns in the resulting `table1` are:

```
['SNR_ID', 'ratio', 'feii']
```

6.3.5 Changing the name of a column

WARNING: There is a pyfits `columns` method called `change_name` but it does not seem to be working properly.

```
table1.columns[table1.columns.names.index('feii')].name='ironII'
table1 = pf.new_table(table1.columns)

table1.columns.names
```

To change the name of a column, one needs to change the `name` attribute of the column. On the first line, the position index of the column named `feii` is used to select the column to change, and then the name of that column is changed to `ironII`.

Again, a *new* table needs to be created once the modifications to the columns are completed.

The `table1` columns are now:

```
['SNR_ID', 'ratio', 'ironII']
```

6.3.6 Appending and deleting rows

Appending and deleting rows is uncannily complicated with PyFITS. This is an area where the use `astropy.table` can certainly help. We hope to be able to add astropy-based examples to this manual in the near future. But for now, let us study the PyFITS way.

Disclaimer: This is the way the author figured out how to do the row manipulations. If the reader knows of a better way to do it with PyFITS, please let us know.

Below, we append two new entries to `table2`. Only the `SNR_ID` and `pabeta` fields will be added to the table since those are the only two columns in `table2`. When an entry has fields not represented in the table, those fields are simply ignored.

```
# New entries for object S004 and S005.
new_entries = {'SNR_ID': ['S004','S005'],
               'ratio' : [1.12, 0.72],
               'feii'  : [77., 87.],
               'pabeta': [69., 122.]
               }
nb_new_entries = len(new_entries['SNR_ID'])

# Create new, larger table.
nrowst2 = table2.data.shape[0]
large_table = pf.new_table(table2.columns, nrows=nrowst2+nb_new_entries)

# Append the new entries and replace table2 with new table.
```

```
14 for name in table2.columns.names:
15    large_table.data.field(name)[nrowst2:] = new_entries[name]
16
17 table2 = large_table
18 print table2.data
```

The values must be entered for each column separately. On Lines 14-15, we loop through the columns by name. To simplify things, it is convenient to have the new values stored in a dictionary keyed on the column names (Lines 2-6).

Adding, and deleting rows (next example), requires the creation of a new table of the correct, new size (Lines 10-11).

```
1  # Delete the last 2 entries from table2
2
3  # Create new, smaller table.
4  nb_bad_entries = 2
5  nrowst2 = table2.data.shape[0]
6  small_table = pf.new_table(table2.columns, nrows=nrowst2-nb_bad_entries)
7
8  # Copy the large table minus the last two lines to the small table.
9  for name in table2.columns.names:
10    small_table.data.field(name)[:] = table2.data.field(name)[:-nb_bad_entries]
11
12 table2 = small_table
```

6.3.7 Changing a value

Changing a value is simply a matter of identifying the column and the row that needs the new value.

Below we show how one might search one column to identify the row and then change that row in another column.

```
# Change the 'pabeta' value for source S002 in table2
rowindex = np.where(table2.data.field('SNR_ID') == 'S002')[0][0]
table2.data.field('pabeta')[rowindex] = 888.
```

APPENDIX

A

LIST OF DESCRIPTORS

A.1 astrodata_FITS

Descriptor	Short Definition	Python type
instrument	name of the instrument	str
object	name of the target	str
telescope	name of the telescope	str
ut_date	UT date	datetime

A.2 astrodata_Gemini

Descriptor	Short Definition	Python type
airmass	airmass of the observation	float
amp_read_area	combination of amplifier name and 1-indexed section relative to the detector.	str
array_name	name assigned to the array generated by a given amplifier, one array per amplifier.	str
array_section	section covered by the array(s), in 0-indexed pixels, relative to the detector frame (e.g. position of multiple amps read within a CCD). List of (x1, x2, y1, y2, ...) tuples	list
azimuth	pointing position in azimuth	float
camera	name of the camera	str
cass_rotator_pa	position angle of the Cassegrain rotator	float
central_wavelength	central wavelength	float
coadds	number of co-adds	int
data_label	Gemini data label	str
data_section	section where the sky-exposed data falls, in 0-indexed pixels. List of (x1, x2, y1, y2, ...) tuples	list
dec	Declination	float
decker	name of the decker	str
detector_name	name assigned to the detector	str
detector_roi_setting	human readable Region Of Interest setting	str
detector_rois_requested	section defining the Regions Of Interest, in 0-indexed pixels. List of (x1, x2, y1, y2, ...) tuples	list
detector_section	section covered by the detector(s), in 0-indexed pixels, relative to the whole mosaic of detectors. List of (x1, x2, y1, y2, ...) tuples	list

Continued on next page

Table A.1 – continued from previous page

Descriptor	Short Definition	Python type
detector_x_bin	X-axis binning	int
detector_y_bin	Y-axis binning	int
disperser	name of the disperser	str
dispersion	value for the dispersion	float
dispersion_axis	dispersion axis	int
elevation	pointing position in elevation	float
exposure_time	exposure time	float
filter_name	name of the effective filter	str
focal_plane_mask	name of the mask in the focal plane	str
gain	gain in electrons per ADU	float
gain_setting	human readable gain setting (eg. low, high)	str
grating	name of the grating	str
group_id	Gemini observation group ID	str
local_time	local time	datetime
lyot_stop	name of the lyot stop	str
mdf_row_id	MDF row ID of a cut MOS spectrum	int
nod_count	number of nods	int
nod_pixels	nod offset in pixels	int
nominal_atmospheric_extinction	nomimal atmospheric extinction	float
nominal_photometric_zeropoint	nominal photometric zeropoint	float
non_linear_level	lower boundary of the non-linear regime	int
observation_class	class of observation (eg. calibration, science)	str
observation_epoch	epoch	str
observation_id	Gemini observation ID	str
observation_type	Gemini observation type	str
overscan_section	section where the overscan data falls, in 0-indexed pixels. List of (x1, x2, y1, y2, ...) tuples	list
pixel_scale	pixel scale in arcsec per pixel	float
prism	name of the prism	str
program_id	Gemini program ID	str
pupil_mask	name of the pupil mask	str
qa_state	quality assessment state (eg. pass, usable, fail)	str
ra	right ascension, in degrees	float
raw_bg	Gemini sky background band	int
raw_cc	Gemini cloud coverage band	int
raw_iq	Gemini image quality band	int
raw_wv	Gemini water vapor band	int
read_mode	Gemini name for combination for gain setting and read setting	str
read_noise	read noise in electrons	float
read_speed_setting	human readable read mode setting (eg. slow, fast)	str
requested_bg	PI requested Gemini sky background band	int
requested_cc	PI requested Gemini cloud coverage band	int
requested_iq	PI requested Gemini image quality band	int
requested_wv	PI requested Gemini water vapor band	int
saturation_level	saturation level	int
slit	name of the slit	str
ut_datetime	UT date and time of the observation	datetime
ut_time	UT time of the observation	datetime
wavefront_sensor	wavefront sensor used for the observation	str

Continued on next page

Table A.1 – continued from previous page

Descriptor	Short Definition	Python type
wavelength_band	band associated with the filter or the central wavelength	str
wavelength_reference_pixel	pixel associated with the central wavelength	float
well_depth_setting	human readable well depth setting (eg. shallow, deep)	str
x_offset	X-axis offset relative to initial pointing position	float
y_offset	Y-axis offset relative to initial pointing position	float

Todo

Write examples that use the DQ plane. Eg. transform DQ plane in a numpy mask and do statistics.

(The *original entry* is located in /data/eclipse/workspace/gemini_python/astrodata/doc/ad_UsersManual/data.rst, line 579.)

Todo

Primer on Descriptor definitions for new instrument.

(The *original entry* is located in /data/eclipse/workspace/gemini_python/astrodata/doc/ad_UsersManual/headers.rst, line 279.)

Todo

Primer on creating new AstroDataTypes.

(The *original entry* is located in /data/eclipse/workspace/gemini_python/astrodata/doc/ad_UsersManual/types.rst, line 152.)