GEMINI

Exploring the Universe, Sharing its Wonders

Astrodata User Manual

Release 3.0.4

Kathleen Labrie

November 2022

Contents

Document ID

PIPE-USER-106_AstrodataUserManual

CHAPTER 1

Introduction

This is the AstroData User's Manual. AstroData is a DRAGONS package. The current chapter covers basic concepts like what is the `astrodata` package and how to install it (together with the other DRAGONS' packages). *Chapter 2* explains with more details what is `AstroData` and how the data is represented using it. *Chapter 3* describes input and output operations and how multi-extension (MEF) FITS files are represented. *Chapter 4* provides information regarding the `TagSet` class, its usage and a few advanced topics. In *Chapter 5* you will find information about the FITS headers and how to access/modify the metadata. The last two chapters, *Chapter 6* and *Chapter 7* cover more details about how to read, manipulate and write pixel data and tables, respectively.

If you are looking for a quick reference, please, have a look on the AstroData Cheat Sheet.

1.1 Reference Documents

- DRAGONS Documentation
- AstroData Cheat Sheet
- Recipe System Users Manual
- Recipe System Programmers Manual

1.2 What is `astrodata`?

`astrodata` is a package that wraps together tools to represent internally astronomical datasets stored on disks and to properly parse their metadata using the `AstroData` and the `TagSet` classes. `astrodata` provides uniform interfaces for working on datasets from different instruments. Once a dataset has been opened with `astrodata`, the object "knows about itself". Information like instrument, observation mode, and how to access headers, is readily available through the uniform interface. All the details are coded inside the class associated with the instrument, that class then provides the interface. The appropriate class is selected automatically when the file is opened and inspected by `astrodata`.

Currently `astrodata` implements a representation for Multi-Extension FITS (MEF) files. (Other representations can be implemented.)

1.3 Installing Astrodata

The `astrodata` package has a few dependencies, astropy, `numpy` and others. The best way to get everything you need is to install Anaconda, and the `DRAGONS` stack from the AstroConda channel.

`astrodata` itself is part of `DRAGONS`. It is available from the repository, as a tar file, or as a conda package. The bare `astrodata` package does not do much by itself, it needs a companion instrument definitions package. For Gemini, this is `gemini_instruments`, also included in `DRAGONS`.

1.3.1 Installing Anaconda and the DRAGONS stack

This is required whether you are installing `DRAGONS` from the repository, the tar file or the conda package.

1. **Install Anaconda.** Go to https://www.anaconda.com/download/ and install the latest 64-bit Anaconda.
2. **Open a bash session.** Anaconda requires bash. If you are not familiar with bash, note that the shell configuration files are named `.bash_profile` and `.bashrc`. During the installation, a PATH setting has been added to your `.bash_profile` or `.bashrc` to add the Anaconda bin directory to the `PATH`.
3. **Activate Anaconda.** Normal Python 3 installation puts the software in `~/anaconda3/`.:

   ```
   $ conda init
   ```

4. **Set up conda channels.** Configure the `conda` package manager to look in the AstroConda channel hosted by STScI, and in the GEMINI Conda Channel. This is a one-time step. It affects current and future Anaconda installations belonging to the same user on the same machine.:

   ```
   $ conda config --add channels http://ssb.stsci.edu/astroconda
   $ conda config --add channels http://astroconda.gemini.edu/public
   ```

5. **Create an environment.** To keep things clean, Anaconda offers virtual environments. Each project can use its own environment. For example, if you do not want to modify the software packages needed for a previous project, just create a new environment for the new project.

   Here we set up an environment where the `DRAGONS` dependencies can be installed without affecting the rest of the system when not using that virtual environement. The new virtual environment here is named `dragons`. The software has been tested with Python 3.7 hence we recommend that you use this specific version of Python with DRAGONS.

   ```
   $ conda create -n dragons python=3.7 dragons stsci
   ```

6. **Activate your new virtual environment.**

   ```
   $ conda activate dragons
   ```

7. **Configure DRAGONS.** These configurations are not stricktly required when using only `astrodata`. It however likely that if you are using `astrodata` you will be using DRAGONS too at some point. So let's configure it to have it ready to go.

   DRAGONS requires a configuration file located in `~/.geminidr/`. The `rsys.cfg` file contains basic configuration for DRAGONS local calibration manager used by `reduce`.

```
$ cd ~
$ mkdir .geminidr
$ cd .geminidr
$ touch rsys.cfg
```

Open `rsys.cfg` with your favorite editor and add these lines:

```
[calibs]
standalone = True
database_dir = ~/.geminidr/
```

Next time you start a DRAGONS project, set the `database_dir` to a path of your liking, this is where the local calibration database will be written.

Then configure buffers for `ds9`:

```
$ cd ~/
$ cp $CONDA_PREFIX/lib/python3.7/site-packages/gempy/numdisplay/imtoolrc ~/.
↪imtoolrc
$ vi .bash_profile (or use your favority editor)
     Add this line to the .bash_profile:
        export IMTOOLRC=~/.imtoolrc
```

1.3.2 Update an existing DRAGONS installation

To check for newer version:

```
$ conda search dragons

The * will show which version is installed if multiple packages are available.
```

To update to the newest version:

```
$ conda update dragons
```

1.3.3 Smoke test the Astrodata installation

From the configured bash shell:

```
$ type python
python is hashed (<home_path>/anaconda3/envs/dragons/python)

Make sure that python is indeed pointing to the Anaconda environment you
have just set up.
```

```
$ python
>>> import astrodata
>>> import gemini_instruments

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

1.3.4 Source code availability

The source code is available on Github:

> https://github.com/GeminiDRSoftware/DRAGONS

1.4 Try it yourself

Try it yourself

Download the data package if you wish to follow along and run the examples presented in this manual. It is available at:

> http://www.gemini.edu/sciops/data/software/datapkgs/ad_usermanual_datapkg-v1.tar

Unpack it:

```
$ cd <somewhere_convenient>
$ tar xvf ad_usermanual_datapkg-v1.tar
$ bunzip2 ad_usermanual/playdata/*.bz2
```

Then

```
$ cd ad_usermanual/playground
$ python
```

1.5 Astrodata Support

Astrodata is developed and supported by staff at the Gemini Observatory. Questions about the reduction of Gemini data should be directed to the Gemini Helpdesk system at `https://www.gemini.edu/sciops/helpdesk/` The github issue tracker can be used to report software bugs in DRAGONS.

CHAPTER 2

The AstroData Object

The `AstroData` object is an internal representation of a file on disk. As of this version, only a FITS layer has been written, but `AstroData` itself is not limited to FITS.

The internal structure of the `AstroData` object makes uses of `astropy.nddata.NDData`, `astropy.table`, and `astropy.io.fits.Header`, the latter simply because it is a convenient ordered dictionary.

Try it yourself

Download the data package (*Try it yourself*) if you wish to follow along and run the examples. Then

```
$ cd <path>/ad_usermanual/playground
$ python
```

2.1 Global vs Extension-specific

At the very top level, the structure is divided in two types of information. In the first category, there is the information that applies to the data globally, for example the information that would be stored in a FITS Primary Header Unit, a table from a catalog that matches the RA and DEC of the field, etc. In the second category, there is the information specific to individual science pixel extensions, for example the gain of the amplifier, the data themselves, the error on those data, etc.

Let us look at an example. The `info()` method shows the content of the `AstroData` object and its organization, from the user's perspective.:

```
>>> import astrodata
>>> import gemini_instruments

>>> ad = astrodata.open('../playdata/N20170609S0154_varAdded.fits')
>>> ad.info()
Filename: N20170609S0154_varAdded.fits
Tags: ACQUISITION GEMINI GMOS IMAGE NORTH OVERSCAN_SUBTRACTED OVERSCAN_TRIMMED
    PREPARED SIDEREAL
```

(continues on next page)

(continued from previous page)

```
Pixels Extensions
Index  Content                  Type              Dimensions     Format
[ 0]   science                  NDAstroData       (2112, 256)    float32
          .variance             ndarray           (2112, 256)    float32
          .mask                 ndarray           (2112, 256)    uint16
          .OBJCAT               Table             (6, 43)        n/a
          .OBJMASK              ndarray           (2112, 256)    uint8
[ 1]   science                  NDAstroData       (2112, 256)    float32
          .variance             ndarray           (2112, 256)    float32
          .mask                 ndarray           (2112, 256)    uint16
          .OBJCAT               Table             (8, 43)        n/a
          .OBJMASK              ndarray           (2112, 256)    uint8
[ 2]   science                  NDAstroData       (2112, 256)    float32
          .variance             ndarray           (2112, 256)    float32
          .mask                 ndarray           (2112, 256)    uint16
          .OBJCAT               Table             (7, 43)        n/a
          .OBJMASK              ndarray           (2112, 256)    uint8
[ 3]   science                  NDAstroData       (2112, 256)    float32
          .variance             ndarray           (2112, 256)    float32
          .mask                 ndarray           (2112, 256)    uint16
          .OBJCAT               Table             (5, 43)        n/a
          .OBJMASK              ndarray           (2112, 256)    uint8

Other Extensions
               Type        Dimensions
.REFCAT        Table       (245, 16)
```

The "Pixel Extensions" contain the pixel data. Each extension is represented individually in a list (0-indexed like all Python lists). The science pixel data, its associated metadata (extension header), and any other pixel or table extensions directly associated with that science pixel data are stored in a `NDAstroData` object which is a subclass of astropy `NDData`. We will return to this structure later. An `AstroData` extension is accessed like any list: `ad[0]`. To access the science pixels, one uses `ad[0].data`; for the object mask of the first extension, `ad[0].OBJMASK`.

In the example above, the "Other Extensions" at the bottom of the `info()` display contains a `REFCAT` table which in this case is a list of stars from a catalog that overlaps the field of view covered by the pixel data. The "Other Extensions" are global extensions. They are not attached to any pixel extension in particular. To access a global extension one simply uses the name of that extension: `ad.REFCAT`.

2.2 Organization of the Global Information

All the global information is stored in attributes of the `AstroData` object. The global headers, or Primary Header Unit (PHU), is stored in the `phu` attribute as an `astropy.io.fits.Header`.

Any global tables, like `REFCAT` above, are stored in the private attribute `_tables` as a Python dictionary with the name (eg. "REFCAT") as the key. All tables are stored as `astropy.table.Table`. Access to those table is done using the key directly as if it were a normal attribute, eg. `ad.REFCAT`. Header information for the table, if read in from a FITS table, is stored in the `meta` attribute of the `astropy.table.Table`, eg. `ad.REFCAT.meta['header']`. It is for information only, it is not used.

2.3 Organization of the Extension-specific Information

The pixel data are stored in the `AstroData` attribute `nddata` as a list of `NDAstroData` object. The `NDAstroData` object is a subclass of astropy `NDData` and it is fully compatible with any function expecting an `NDData` as input. The pixel extensions are accessible through slicing, eg. `ad[0]` or even `ad[0:2]`. A slice of an AstroData object is an AstroData object, and all the global attributes are kept. For example:

```
>>> ad[0].info()
Filename: N20170609S0154_varAdded.fits
Tags: ACQUISITION GEMINI GMOS IMAGE NORTH OVERSCAN_SUBTRACTED OVERSCAN_TRIMMED
    PREPARED SIDEREAL

Pixels Extensions
Index  Content                  Type              Dimensions     Format
[ 0]   science                  NDAstroData       (2112, 256)    float32
          .variance             ndarray           (2112, 256)    float32
          .mask                 ndarray           (2112, 256)    uint16
          .OBJCAT               Table             (6, 43)        n/a
          .OBJMASK              ndarray           (2112, 256)    uint8

Other Extensions
               Type        Dimensions
.REFCAT        Table       (245, 16)
```

Note how `REFCAT` is still present.

The science data is accessed as `ad[0].data`, the variance as `ad[0].variance`, and the data quality plane as `ad[0].mask`. Those familiar with astropy `NDData` will recognize the structure "data, error, mask", and will notice some differences. First `AstroData` uses the variance for the error plane, not the standard deviation. Another difference will be evident only when one looks at the content of the mask. `NDData` masks contain booleans, `AstroData` masks are `uint16` bit mask that contains information about the type of bad pixels rather than just flagging them a bad or not. Since `0` is equivalent to `False` (good pixel), the `AstroData` mask is fully compatible with the `NDData` mask.

Header information for the extension is stored in the `NDAstroData` `meta` attribute. All table and pixel extensions directly associated with the science extension are also stored in the `meta` attribute.

Technically, an extension header is located in `ad.nddata[0].meta['header']`. However, for obviously needed convenience, the normal way to access that header is `ad[0].hdr`.

Tables and pixel arrays associated with a science extension are stored in `ad.nddata[0].meta['other']` as a dictionary keyed on the array name, eg. `OBJCAT`, `OBJMASK`. As it is for global tables, astropy tables are used for extension tables. The extension tables and extra pixel arrays are accessed, like the global tables, by using the table name rather than the long format, for example `ad[0].OBJCAT` and `ad[0].OBJMASK`.

When reading FITS Table, the header information is stored in the `meta['header']` of the table, eg. `ad[0].OBJCAT.meta['header']`. That information is not used, it is simply a place to store what was read from disk.

The header of a pixel extension directly associated with the science extension should match that of the science extension. Therefore such headers are not stored in `AstroData`. For example, the header of `ad[0].OBJMASK` is the same as that of the science, `ad[0].hdr`.

2.4 A Note on Memory Usage

When an file is opened, the headers are loaded into memory, but the pixels are not. The pixel data are loaded into memory only when they are first needed. This is not real "memory mapping", more of a delayed loading. This is useful when someone is only interested in the metadata, especially when the files are very large.

CHAPTER 3

Input and Output Operations and Extension Manipulation - MEF

AstroData is not intended to be Multi-Extension FITS (MEF) centric. The core is independent of the file format. At Gemini, our data model uses MEF. Therefore we have implemented a FITS handler that maps a MEF to the internal AstroData representation. A different handler can be implemented for a different file format.

In this chapter, we present examples that will help the reader understand how to access the information stored in a MEF with the AstroData object and understand that mapping.

Try it yourself

Download the data package (*Try it yourself*) if you wish to follow along and run the examples. Then

```
$ cd <path>/ad_usermanual/playground
$ python
```

3.1 Imports

Before doing anything, you need to import AstroData and the Gemini instrument configuration package gemini_instruments.

```
>>> import astrodata
>>> import gemini_instruments
```

3.2 Open and access existing dataset

3.2.1 Read in the dataset

The file on disk is loaded into the AstroData class associated with the instrument the data is from. This association is done automatically based on header content.

```
>>> ad = astrodata.open('../playdata/N20170609S0154.fits')
>>> type(ad)
<class 'gemini_instruments.gmos.adclass.AstroDataGmos'>
```

From now on, `ad` knows it is GMOS data. It knows how to access its headers and when using the Recipe System (`recipe_system`), it will trigger the selection of the GMOS primitives and recipes.

The original path and filename are stored in the object. If you were to write the `AstroData` object to disk without specifying anything, those path and filename would be used.

```
>>> ad.path
'../playdata/N20170609S0154.fits'
>>> ad.filename
'N20170609S0154.fits'
```

3.2.2 Accessing the content of a MEF file

Accessing pixel data, headers, and tables will be covered in details in the following chapters. Here we just introduce the basic content interface.

For details on the `AstroData` structure, please refer to the *previous chapter*.

`AstroData` uses `NDData` as the core of its structure. Each FITS extension becomes a `NDAstroData` object, subclassed from `NDData`, and is added to a list.

Pixel data

To access pixel data, the list index and the `.data` attribute are used. That returns a `numpy.ndarray`. The list of `NDAstroData` is zero-indexed. *Extension number 1 in a MEF is index 0 in an |AstroData| object.*

```
>>> ad = astrodata.open('../playdata/N20170609S0154_varAdded.fits')
>>> data = ad[0].data
>>> type(data)
<type 'numpy.ndarray'>
>>> data.shape
(2112, 256)
```

Remember that in a `ndarray` the y-axis is the first number.

The variance and data quality planes, the VAR and DQ planes in Gemini MEF files, are represented by the `.variance` and `.mask` attributes, respectively. They are not their own "extension", they don't have their own index in the list, unlike in a MEF. They are attached to the pixel data, packaged together by the `NDAstroData` object. They are represented as `numpy.ndarray` just like the pixel data

```
>>> var = ad[0].variance
>>> dq = ad[0].mask
```

Tables

Tables in the MEF file will also be loaded into the `AstroData` object. If a table is associated with a specific science extension through the EXTVER header, that table will be packaged within the same AstroData extension as the pixel data. The `AstroData` "extension" is the `NDAstroData` object plus any table or other pixel array. If the table is not associated with a specific extension and applies globally, it will be added to the AstroData object as a global addition.

No indexing will be required to access it. In the example below, one OBJCAT is associated with each extension, while the REFCAT has a global scope

```
>>> ad.info()
Filename: ../playdata/N20170609S0154_varAdded.fits
Tags: ACQUISITION GEMINI GMOS IMAGE NORTH OVERSCAN_SUBTRACTED OVERSCAN_TRIMMED
    PREPARED SIDEREAL

Pixels Extensions
Index  Content                  Type              Dimensions     Format
[ 0]   science                  NDAstroData       (2112, 256)    float32
          .variance             ndarray           (2112, 256)    float32
          .mask                 ndarray           (2112, 256)    uint16
          .OBJCAT               Table             (6, 43)        n/a
          .OBJMASK              ndarray           (2112, 256)    uint8
[ 1]   science                  NDAstroData       (2112, 256)    float32
          .variance             ndarray           (2112, 256)    float32
          .mask                 ndarray           (2112, 256)    uint16
          .OBJCAT               Table             (8, 43)        n/a
          .OBJMASK              ndarray           (2112, 256)    uint8
[ 2]   science                  NDAstroData       (2112, 256)    float32
          .variance             ndarray           (2112, 256)    float32
          .mask                 ndarray           (2112, 256)    uint16
          .OBJCAT               Table             (7, 43)        n/a
          .OBJMASK              ndarray           (2112, 256)    uint8
[ 3]   science                  NDAstroData       (2112, 256)    float32
          .variance             ndarray           (2112, 256)    float32
          .mask                 ndarray           (2112, 256)    uint16
          .OBJCAT               Table             (5, 43)        n/a
          .OBJMASK              ndarray           (2112, 256)    uint8

Other Extensions
               Type        Dimensions
.REFCAT        Table       (245, 16)
```

The tables are stored internally as astropy.table.Table objects.

```
>>> ad[0].OBJCAT
<Table length=6>
NUMBER X_IMAGE Y_IMAGE ... REF_MAG_ERR PROFILE_FWHM PROFILE_EE50
int32  float32 float32 ...   float32      float32      float32
------ ------- ------- ... ----------- ------------ ------------
     1 283.461 55.4393 ...     0.16895       -999.0       -999.0
...
>>> type(ad[0].OBJCAT)
<class 'astropy.table.table.Table'>

>>> refcat = ad.REFCAT
>>> type(refcat)
<class 'astropy.table.table.Table'>
```

Headers

Headers are stored in the NDAstroData .meta attribute as astropy.io.fits.Header objects, which is a form of Python ordered dictionaries. Headers associated with extensions are stored with the corresponding NDAstroData object. The MEF Primary Header Unit (PHU) is stored “globally” in the AstroData object. Note that when slicing an AstroData object, for example copying over just the first extension, the PHU will follow. The slice of an `AstroData` object is an `AstroData` object. Headers can be accessed directly, or for some predefined concepts, the use of Descriptors is preferred. See the chapters on headers for details.

Using Descriptors:

```
>>> ad = astrodata.open('../playdata/N20170609S0154.fits')
>>> ad.filter_name()
'open1-6&g_G0301'
>>> ad.filter_name(pretty=True)
'g'
```

Using direct header access:

```
>>> ad.phu['FILTER1']
'open1-6'
>>> ad.phu['FILTER2']
'g_G0301'
```

Accessing the extension headers:

```
>>> ad.hdr['CCDSEC']
['[1:512,1:4224]', '[513:1024,1:4224]', '[1025:1536,1:4224]', '[1537:2048,1:4224]']
>>> ad[0].hdr['CCDSEC']
'[1:512,1:4224]'

With descriptors:
>>> ad.array_section(pretty=True)
['[1:512,1:4224]', '[513:1024,1:4224]', '[1025:1536,1:4224]', '[1537:2048,1:4224]']
```

3.3 Modify Existing MEF Files

Before you start modify the structure of an `AstroData` object, you should be familiar with it. Please make sure that you have read the previous chapter on *the structure of the AstroData object*.

3.3.1 Appending an extension

In this section, we take an extension from one `AstroData` object and append it to another. Because we are mapping a FITS file, the `EXTVER` keyword gets automatically updated to the next available value to ensure that when the `AstroData` object is written back to disk as MEF, it will be coherent.

Here is an example appending a whole AstroData extension, with pixel data, variance, mask and tables.

```
>>> ad = astrodata.open('../playdata/N20170609S0154.fits')
>>> advar = astrodata.open('../playdata/N20170609S0154_varAdded.fits')

>>> ad.info()
Filename: ../playdata/N20170609S0154.fits
Tags: ACQUISITION GEMINI GMOS IMAGE NORTH RAW SIDEREAL UNPREPARED
Pixels Extensions
Index  Content                 Type              Dimensions     Format
[ 0]   science                 NDAstroData       (2112, 288)    uint16
[ 1]   science                 NDAstroData       (2112, 288)    uint16
[ 2]   science                 NDAstroData       (2112, 288)    uint16
[ 3]   science                 NDAstroData       (2112, 288)    uint16
```

(continues on next page)

(continued from previous page)

```
>>> ad.append(advar[3])
>>> ad.info()
Filename: ../playdata/N20170609S0154.fits
Tags: ACQUISITION GEMINI GMOS IMAGE NORTH RAW SIDEREAL UNPREPARED
Pixels Extensions
Index  Content                  Type             Dimensions     Format
[ 0]   science                  NDAstroData      (2112, 288)    uint16
[ 1]   science                  NDAstroData      (2112, 288)    uint16
[ 2]   science                  NDAstroData      (2112, 288)    uint16
[ 3]   science                  NDAstroData      (2112, 288)    uint16
[ 4]   science                  NDAstroData      (2112, 256)    float32
          .variance             ndarray          (2112, 256)    float32
          .mask                 ndarray          (2112, 256)    int16
          .OBJCAT               Table            (5, 43)        n/a
          .OBJMASK              ndarray          (2112, 256)    uint8

>>> ad[4].hdr['EXTVER']
5
>>> advar[3].hdr['EXTVER']
4
```

As you can see above, the fourth extension of `advar`, along with everything it contains was appended at the end of the first `AstroData` object. Also, note that the EXTVER of the extension in `advar` was 4, but once appended to `ad`, it had to be changed to the next available integer, 5, numbers 1 to 4 being already used by `ad`'s own extensions.

In this next example, we are appending only the pixel data, leaving behind the other associated data. One can attach the headers too, like we do here.

```
>>> ad = astrodata.open('../playdata/N20170609S0154.fits')
>>> advar = astrodata.open('../playdata/N20170609S0154_varAdded.fits')

>>> ad.append(advar[3].data, header=advar[3].hdr)
>>> ad.info()
Filename: ../playdata/N20170609S0154.fits
Tags: ACQUISITION GEMINI GMOS IMAGE NORTH RAW SIDEREAL UNPREPARED
Pixels Extensions
Index  Content                  Type             Dimensions     Format
[ 0]   science                  NDAstroData      (2112, 288)    uint16
[ 1]   science                  NDAstroData      (2112, 288)    uint16
[ 2]   science                  NDAstroData      (2112, 288)    uint16
[ 3]   science                  NDAstroData      (2112, 288)    uint16
[ 4]   science                  NDAstroData      (2112, 256)    float32
```

Notice how a new extension was created but `variance`, `mask`, the OBJCAT table and OBJMASK image were not copied over. Only the science pixel data was copied over.

Please note, there is no implementation for the "insertion" of an extension.

3.3.2 Removing an extension or part of one

Removing an extension or a part of an extension is straightforward. The Python command `del()` is used on the item to remove. Below are a few examples, but first let us load a file

```
>>> ad = astrodata.open('../playdata/N20170609S0154_varAdded.fits')
>>> ad.info()
```

As you go through these examples, check the new structure with `ad.info()` after every removal to see how the structure has changed.

Deleting a whole `AstroData` extension, the fourth one

```
>>> del ad[3]
```

Deleting only the variance array from the second extension

```
>>> ad[1].variance = None
```

Deleting a table associated with the first extension

```
>>> del ad[0].OBJCAT
```

Deleting a global table, not attached to a specific extension

```
>>> del ad.REFCAT
```

3.4 Writing back to disk

The `AstroDataFits` layer takes care of converting the `AstroData` object back to a MEF file on disk. When writing to disk, one should be aware of the path and filename information associated with the `AstroData` object.

```
>>> ad = astrodata.open('../playdata/N20170609S0154.fits')
>>> ad.path
'../playdata/N20170609S0154.fits'
>>> ad.filename
'N20170609S0154.fits'
```

3.4.1 Writing to a new file

There are various ways to define the destination for the new FITS file. The most common and natural way is

```
>>> ad.write('new154.fits')

>>> ad.write('new154.fits', overwrite=True)
```

This will write a FITS file named ‘new154.fits’ in the current directory. With `overwrite=True`, it will overwrite the file if it already exists. A path can be prepended to the filename if the current directory is not the destination. Note that `ad.filename` and `ad.path` have not changed, we have just written to the new file, the `AstroData` object is in no way associated with that new file.

```
>>> ad.path
'../playdata/N20170609S0154.fits'
>>> ad.filename
'N20170609S0154.fits'
```

If you want to create that association, the `ad.filename` and `ad.path` needs to be modified first. For example:

```
>>> ad.filename = 'new154.fits'
>>> ad.write(overwrite=True)

```

(continues on next page)

(continued from previous page)

```
>>> ad.path
'../playdata/new154.fits'
>>> ad.filename
'new154.fits'
```

Changing `ad.filename` also changes the filename in the `ad.path`. The sequence above will write 'new154.fits' not in the current directory but rather to the directory that is specified in `ad.path`.

WARNING: `ad.write()` has an argument named `filename`. Setting `filename` in the call to `ad.write()`, as in `ad.write(filename='new154.fits')` will NOT modify `ad.filename` or `ad.path`. The two "filenames", one a method argument the other a class attribute have no association to each other.

3.4.2 Updating an existing file on disk

Updating an existing file on disk requires explicitly allowing overwrite.

If you have not written 'new154.fits' to disk yet (from previous section)

```
>>> ad = astrodata.open('../playdata/N20170609S0154.fits')
>>> ad.write('new154.fits', overwrite=True)
```

Now let's open 'new154.fits', and write to it

```
>>> adnew = astrodata.open('new154.fits')
>>> adnew.write(overwrite=True)
```

3.5 Create New MEF Files

A new MEF file can be created from an existing, maybe modified, file or it can be created from scratch. We discuss both cases here.

3.5.1 Create New Copy of MEF Files

To create a new copy of a MEF file, modified or not, the user has already been given most of the tools in the sections above. Yet, let's throw a couple examples for completeness.

Basic example

As seen above, a MEF file can be opened with `astrodata`, the `AstroData` object can be modified (or not), and then written back to disk under a new name.

```
>>> ad = astrodata.open('../playdata/N20170609S0154.fits')
... optional modifications here ...
>>> ad.write('newcopy.fits')
```

Needing true copies in memory

Sometimes it is a true copy in memory that is needed. This is not specific to MEF. In Python, doing something like `adnew = ad` does not create a new copy of the AstrodData object; it just gives it a new name. If you modify `adnew` you will be modifying `ad` too. They point to the same block of memory.

To create a true independent copy, the deepcopy utility needs to be used.

```
>>> from copy import deepcopy
>>> ad = astrodata.open('../playdata/N20170609S0154.fits')
>>> adcopy = deepcopy(ad)
```

Be careful using deepcopy, your memory could balloon really fast. Use it only when truly needed.

3.5.2 Create New MEF Files from Scratch

Before one creates a new MEF file on disk, one has to create the AstroData object that will be eventually written to disk. The AstroData object created also needs to know that it will have to be written using the MEF format. This is fortunately handled fairly transparently by astrodata.

The key to associating the FITS data provider to the AstroData object is simply to create the AstroData object from astropy.io.fits header objects. Those will be recognized by astrodata as FITS and the constructor for FITS will be used. The user does not need to do anything else special. Here is how it is done.

Create a MEF with basic header and data array set to zeros

```
>>> import numpy as np
>>> from astropy.io import fits

>>> phu = fits.PrimaryHDU()

>>> pixel_data = np.zeros((100,100))

>>> hdu = fits.ImageHDU()
>>> hdu.data = pixel_data

>>> ad = astrodata.create(phu)
>>> ad.append(hdu, name='SCI')

or another way to do the last two blocs:
>>> hdu = fits.ImageHDU(data=pixel_data, name='SCI')
>>> ad = astrodata.create(phu, [hdu])
```

Then it is just a matter of calling ad.write('somename.fits') on that new Astrodata object.

Represent a table as a FITS binary table in an AstroData object

One first needs to create a table, either an astropy.table.Table or a BinTableHDU. See the astropy documentation on tables and this manual's *section* dedicated to tables for more information.

In the first example, we assume that my_astropy_table is a Table ready to be attached to an AstroData object. (Warning: we have not created my_astropy_table therefore the example below will not run, though this is how it would be done.)

```
>>> phu = fits.PrimaryHDU()
>>> ad = astrodata.create(phu)

>>> astrodata.add_header_to_table(my_astropy_table)
>>> ad.append(my_astropy_table, name='SMAUG')
```

In the second example, we start with a FITS `BinTableHDU` and attach it to a new `AstroData` object. (Again, we have not created `my_fits_table` so the example will not run.)

```
>>> phu = fits.PrimaryHDU()
>>> ad = astrodata.create(phu)
>>> ad.append(my_fits_table, name='DROGON')
```

As before, once the `AstroData` object is constructed, the `ad.write()` method can be used to write it to disk as a MEF file.

CHAPTER 4

Astrodata Tags

4.1 What are the Astrodata Tags?

The Astrodata Tags identify the data represented in the `AstroData` object. When a file on disk is opened with `astrodata`, the headers are inspected to identify which specific `AstroData` class needs to be loaded, `AstroDataGmos`, `AstroDataNiri`, etc. Based on the class the data is associated with, a list of "tags" will be defined. The tags will tell whether the file is a flatfield or a dark, if it is a raw dataset, or if it has been processed by the recipe system, if it is imaging or spectroscopy. The tags will tell the users and the system what that data is and also give some information about the processing status.

As a side note, the tags are used by DRAGONS Recipe System to match recipes and primitives to the data.

4.2 Using the Astrodata Tags

Try it yourself

Download the data package (*Try it yourself*) if you wish to follow along and run the examples. Then

```
$ cd <path>/ad_usermanual/playground
$ python
```

Before doing anything, you need to import `astrodata` and the Gemini instrument configuration package (`gemini_instruments`).

```
>>> import astrodata
>>> import gemini_instruments
```

Let us open a Gemini dataset and see what tags we get:

```
>>> ad = astrodata.open('../playdata/N20170609S0154.fits')
>>> ad.tags
{'RAW', 'GMOS', 'GEMINI', 'NORTH', 'SIDEREAL', 'UNPREPARED', 'IMAGE', 'ACQUISITION'}
```

The file we loaded is raw, GMOS North data. It is a 2D image and it is an acquisition image, not a science observation. The "UNPREPARED" tag indicates that the file has never been touched by the Recipe System which runs a "prepare" primitive as the first step of each recipe.

Let's try another

```
>>> ad = astrodata.open('../playdata/N20170521S0925_forStack.fits')
>>> ad.tags
{'GMOS', 'GEMINI', 'NORTH', 'SIDEREAL', 'OVERSCAN_TRIMMED', 'IMAGE',
'OVERSCAN_SUBTRACTED', 'PREPARED'}
```

This file is a science GMOS North image. It has been processed by the Recipe System. The overscan level has been subtracted and the section has been trimmed away. The tags do NOT include all the processing steps. Rather, at least from the time being, it focuses on steps that matter when associating calibrations.

The tags can be used when coding. For example:

```
>>> if 'GMOS' in ad.tags:
...     print('I am GMOS')
... else:
...     print('I am these instead:', ad.tags)
...
```

And:

```
>>> if {'IMAGE', 'GMOS'}.issubset(ad.tags):
...     print('I am a GMOS Image.')
...
```

4.3 Using typewalk

In DRAGONS, there is a convenience tool that will list the Astrodata tags for all the FITS file in a directory.

To try it, from the shell, not Python, go to the "playdata" directory and run typewalk:

```
% cd <path>/ad_usermanual/playdata
% typewalk

directory:  /data/workspace/ad_usermanual/playdata
 N20170521S0925_forStack.fits ...... (GEMINI) (GMOS) (IMAGE) (NORTH) (OVERSCAN_
↪SUBTRACTED) (OVERSCAN_TRIMMED) (PREPARED) (SIDEREAL)
 N20170521S0926_forStack.fits ...... (GEMINI) (GMOS) (IMAGE) (NORTH) (OVERSCAN_
↪SUBTRACTED) (OVERSCAN_TRIMMED) (PREPARED) (PROCESSED) (PROCESSED_SCIENCE) (SIDEREAL)
 N20170609S0154.fits ............... (ACQUISITION) (GEMINI) (GMOS) (IMAGE) (NORTH)␣
↪(RAW) (SIDEREAL) (UNPREPARED)
 N20170609S0154_varAdded.fits ...... (ACQUISITION) (GEMINI) (GMOS) (IMAGE) (NORTH)␣
↪(OVERSCAN_SUBTRACTED) (OVERSCAN_TRIMMED) (PREPARED) (SIDEREAL)
 estgsS20080220S0078.fits .......... (GEMINI) (GMOS) (LONGSLIT) (LS) (PREPARED)␣
↪(PROCESSED) (PROCESSED_SCIENCE) (SIDEREAL) (SOUTH) (SPECT)
 gmosifu_cube.fits ................. (GEMINI) (GMOS) (IFU) (NORTH) (ONESLIT_RED)␣
↪(PREPARED) (PROCESSED) (PROCESSED_SCIENCE) (SIDEREAL) (SPECT)
 new154.fits ....................... (ACQUISITION) (GEMINI) (GMOS) (IMAGE) (NORTH)␣
↪(RAW) (SIDEREAL) (UNPREPARED)
Done DataSpider.typewalk(..)
```

`typewalk` can be used to select specific data based on tags, and even create lists:

```
% typewalk --tags RAW
directory:  /data/workspace/ad_usermanual/playdata
 N20170609S0154.fits ............... (ACQUISITION) (GEMINI) (GMOS) (IMAGE) (NORTH)␣
↪(RAW) (SIDEREAL) (UNPREPARED)
 new154.fits ....................... (ACQUISITION) (GEMINI) (GMOS) (IMAGE) (NORTH)␣
↪(RAW) (SIDEREAL) (UNPREPARED)
Done DataSpider.typewalk(..)
```

```
% typewalk --tags RAW -o rawfiles.lis
% cat rawfiles.lis
# Auto-generated by typewalk, vv2.0 (beta)
# Written: Tue Mar  6 13:06:06 2018
# Qualifying types: RAW
# Qualifying logic: AND
# -----------------------
/Users/klabrie/data/tutorials/ad_usermanual/playdata/N20170609S0154.fits
/Users/klabrie/data/tutorials/ad_usermanual/playdata/new154.fits
```

4.4 Creating New Astrodata Tags [Advanced Topic]

For proper and complete instructions on how to create Astrodata Tags and the `AstroData` class that hosts the tags, the reader is invited to refer to the Astrodata Programmer Manual. Here we provide a simple introduction that might help some readers better understand Astrodata Tags, or serve as a quick reference for those who have written Astrodata Tags in the past but need a little refresher.

The Astrodata Tags are defined in an `AstroData` class. The `AstroData` class specific to an instrument is located in a separate package, not in `astrodata`. For example, for Gemini instruments, all the various `AstroData` classes are contained in the `gemini_instruments` package.

An Astrodata Tag is a function within the instrument's `AstroData` class. The tag function is distinguished from normal functions by applying the `astro_data_tag()` decorator to it. The tag function returns a `astrodata.TagSet`.

For example:

```
class AstroDataGmos(AstroDataGemini):
    ...
    @astro_data_tag
    def _tag_arc(self):
        if self.phu.get('OBSTYPE) == 'ARC':
            return TagSet(['ARC', 'CAL'])
```

The tag function looks at the headers and if the keyword "OBSTYPE" is set to "ARC", the tags "ARC" and "CAL" (for calibration) will be assigned to the `AstroData` object.

A whole suite of such tag functions is needed to fully characterize all types of data an instrument can produce.

Tags are about what the dataset is, not it's flavor. The Astrodata "descriptors" (see the section on *Metadata and Headers*) will describe the flavor. For example, tags will say that the data is an image, but the descriptor will say whether it is B-band or R-band. Tags are used for recipe and primitive selection. A way to understand the difference between a tag and a descriptor is in terms of the recipe that will be selected: A GMOS image will use the same recipe whether it's a B-band or R-band image. However, a GMOS longslit spectrum will need a very different recipe. A bias is reduced differently from a science image, there should be a tag differentiating a bias from a science image. (There is for GMOS.)

For more information on adding to Astrodata, see the Astrodata Programmer Manual.

CHAPTER 5

Metadata and Headers

Try it yourself

Download the data package (*Try it yourself*) if you wish to follow along and run the examples. Then

```
$ cd <path>/ad_usermanual/playground
$ python
```

You need to import Astrodata and the Gemini instrument configuration package.

```
>>> import astrodata
>>> import gemini_instruments
```

5.1 Astrodata Descriptors

We show in this chapter how to use the Astrodata Descriptors. But first let's explain what they are.

Astrodata Descriptors provide a "header-to-concept" mapping that allows the user to access header information from a unique interface, regardless of which instrument the dataset is from. Like for the Astrodata Tags, the mapping is coded in a configuration package separate from core Astrodata. For Gemini instruments, that package is named `gemini_instruments`.

For example, if the user is interested to know the effective filter used for an observation, normally one needs to know which specific keyword or set of keywords to look at for that instrument. However, once the concept of "filter" is coded as a Descriptor, the user only needs to call the `filter_name()` descriptor to retrieve the information.

The Descriptors are closely associated with the Astrodata Tags. In fact, they are implemented in the same `AstroData` class as the tags. Once the specific `AstroData` class is selected (upon opening the file), all the tags and descriptors for that class are defined. For example, all the descriptor functions of GMOS data, ie. the functions that map a descriptor concept to the actual header content, are defined in the `AstroDataGmos` class.

This is all completely transparent to the user. One simply opens the data file and all the descriptors are ready to be used.

Note: Of course if the Descriptors have not been implemented for that specific data, they will not work. They should all be defined for Gemini data. For other sources, the headers can be accessed directly, one keyword at a time. This type of access is discussed below. This is also useful when the information needed is not associated with one of the standard descriptors.

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

```
>>> ad = astrodata.open('../playdata/N20170609S0154.fits')
>>> ad.descriptors
('airmass', 'amp_read_area', 'ao_seeing', ...
  ...)
```

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)
>>> help(ad.filter_name)
```

The full list of standard descriptors is available in the Appendix *List of Gemini Standard Descriptors*.

5.2 Accessing Metadata

5.2.1 Accessing Metadata with Descriptors

Whenever possible the Descriptors should be used to get information from headers. This allows for maximum re-usability of the code as it will then work on any datasets with an `AstroData` class.

Here are a few examples using Descriptors:

```
>>> ad = astrodata.open('../playdata/N20170609S0154.fits')

>>> #--- print a value
>>> print('The airmass is : ', ad.airmass())

>>> #--- use a value to control the flow
>>> if ad.exposure_time() < 240.:
...     print('This is a short exposure.')
... else:
...     print('This is a long exposure.')
...

>>> #--- multiply all extensions by their respective gain
>>> for ext, gain in zip(ad, ad.gain()):
...     ext *= gain
...

>>> #--- do arithmetics
>>> fwhm_pixel = 3.5
>>> fwhm_arcsec = fwhm_pixel * ad.pixel_scale()
```

The return values for Descriptors depend on the nature of the information being requested and the number of extensions in the `AstroData` object. When the value has words, it will be string, if it is a number it will be a float or an integer. The dataset used in this section has 4 extensions. When the descriptor value can be different for each extension, the descriptor will return a Python list.

```
>>> ad.airmass()
1.089
>>> ad.gain()
[2.03, 1.97, 1.96, 2.01]
>>> ad.filter_name()
'open1-6&g_G0301'
```

Some descriptors accept arguments. For example:

```
>>> ad.filter_name(pretty=True)
'g'
```

A full list of standard descriptors is available in the Appendix *List of Gemini Standard Descriptors*.

5.2.2 Accessing Metadata Directly

Not all header content is mapped to Descriptors, nor should it. Direct access is available for header content falling outside the scope of the descriptors.

One important thing to keep in mind is that the PHU (Primary Header Unit) and the extension headers are accessed slightly differently. The attribute `phu` needs to be used for the PHU, and `hdr` for the extension headers.

Here are some examples of direct header access:

```
>>> ad = astrodata.open('../playdata/N20170609S0154.fits')

>>> #--- Get keyword value from the PHU
>>> ad.phu['AOFOLD']
'park-pos.'

>>> #--- Get keyword value from a specific extension
>>> ad[0].hdr['CRPIX1']
511.862999160781

>>> #--- Get keyword value from all the extensions in one call.
>>> ad.hdr['CRPIX1']
[511.862999160781, 287.862999160781, -0.137000839218696, -224.137000839219]
```

5.2.3 Whole Headers

Entire headers can be retrieved as `fits` `Header` objects:

```
>>> ad = astrodata.open('../playdata/N20170609S0154.fits')
>>> type(ad.phu)
<class 'astropy.io.fits.header.Header'>
>>> type(ad[0].hdr)
<class 'astropy.io.fits.header.Header'>
```

In interactive mode, it is possible to print the headers on the screen as follows:

```
>>> ad.phu
SIMPLE  =                    T / file does conform to FITS standard
BITPIX  =                   16 / number of bits per data pixel
NAXIS   =                    0 / number of data axes
....
```

(continues on next page)

(continued from previous page)

```
>>> ad[0].hdr
XTENSION= 'IMAGE   '           / IMAGE extension
BITPIX  =                   16 / number of bits per data pixel
NAXIS   =                    2 / number of data axes
....
```

5.3 Updating, Adding and Deleting Metadata

Header cards can be updated, added to, or deleted from the headers. The PHU and the extensions headers are again accessed in a mostly identical way with phu and hdr, respectively.

```
>>> ad = astrodata.open('../playdata/N20170609S0154.fits')
```

Add and update a keyword, without and with comment:

```
>>> ad.phu['NEWKEY'] = 50.
>>> ad.phu['NEWKEY'] = (30., 'Updated PHU keyword')

>>> ad[0].hdr['NEWKEY'] = 50.
>>> ad[0].hdr['NEWKEY'] = (30., 'Updated extension keyword')
```

Delete a keyword:

```
>>> del ad.phu['NEWKEY']
>>> del ad[0].hdr['NEWKEY']
```

5.4 World Co-ordinate System attribute

The `wcs` of an extension's `nddata` attribute (eg. `ad[0].nddata.wcs`; see *Pixel Data*) may contain an instance of `astropy.wcs.WCS` (a standard FITS WCS object) or `gwcs.wcs.WCS` (a "Generalized WCS" or gWCS object). These both define a transformation between array indices and some other co-ordinate system such as "World" co-ordinates (see APE 14). GWCS allows multiple, almost arbitrary co-ordinate mappings from different calibration steps (eg. CCD mosaicking, distortion correction & wavelength calibration) to be combined in a single, reversible transformation chain — but this information cannot all be represented as a FITS standard WCS. If a gWCS object is defined here, it gets stored as a table extension named 'WCS' when the `AstroData` instance is saved to a file (with the same EXTVER as the corresponding 'SCI' array). This is independent of any WCS information already stored in the FITS headers. The representation in the table is produced using ASDF, with one line of text per row. Likewise, when the file is re-opened, the gWCS object gets recreated in `wcs` from the table.

In future, it is intended that the `wcs` attribute will get populated from standard FITS headers where there is no overriding 'WCS' table extension and will get saved to standard FITS headers when its type is `astropy.wcs.WCS`. Also, where a gWCS object is used, a discrete sampling of the World co-ordinate values will be stored as part of the FITS WCS, following Greisen et al. (2006), S6 (in addition to the definitive 'WCS' table), allowing standard FITS readers to report accurate World co-ordinates for each pixel.

5.5 Adding Descriptors [Advanced Topic]

For proper and complete instructions on how to create Astrodata Descriptors, the reader is invited to refer to the Astrodata Programmer Manual. Here we provide a simple introduction that might help some readers better understand Astrodata Descriptors, or serve as a quick reference for those who have written Astrodata Descriptors in the past but need a little refresher.

The Astrodata Descriptors are defined in an `AstroData` class. The `AstroData` class specific to an instrument is located in a separate package, not in `astrodata`. For example, for Gemini instruments, all the various `AstroData` classes are contained in the `gemini_instruments` package.

An Astrodata Descriptor is a function within the instrument's `AstroData` class. The descriptor function is distinguished from normal functions by applying the `@astro_data_descriptor` decorator to it. The descriptor function returns the value(s) using a Python type, `int`, `float`, `string`, `list`; it depends on the value being returned. There is no special "descriptor" type.

Here is an example of code defining a descriptor:

```python
class AstroDataGmos(AstroDataGemini):
    ...
    @astro_data_descriptor
    def detector_x_bin(self):
        def _get_xbin(b):
            try:
                return int(b.split()[0])
            except (AttributeError, ValueError):
                return None

        binning = self.hdr.get('CCDSUM')
        if self.is_single:
            return _get_xbin(binning)
        else:
            xbin_list = [_get_xbin(b) for b in binning]
            # Check list is single-valued
            return xbin_list[0] if xbin_list == xbin_list[::-1] else None
```

This descriptor returns the X-axis binning as a integer when called on a single extension, or an object with only one extension, for example after the GMOS CCDs have been mosaiced. If there are more than one extensions, it will return a Python list or an integer if the binning is the same for all the extensions.

Gemini has defined a standard list of descriptors that should be defined one way or another for each instrument to ensure the re-usability of our algorithms. That list is provided in the Appendix *List of Gemini Standard Descriptors*.

For more information on adding to Astrodata, see the Astrodata Programmer Manual.

CHAPTER 6

Pixel Data

Try it yourself

Download the data package (*Try it yourself*) if you wish to follow along and run the examples. Then

```
$ cd <path>/ad_usermanual/playground
$ python
```

Then import core astrodata and the Gemini astrodata configurations.

```
>>> import astrodata
>>> import gemini_instruments
```

6.1 Operate on Pixel Data

The pixel data are stored in the `AstroData` object as a list of `NDAstroData` objects. The `NDAstroData` is a subclass of Astropy's `NDData` class which combines in one "package" the pixel values, the variance, and the data quality plane or mask (as well as associated meta-data). The data can be retrieved as a standard NumPy `ndarray`.

In the sections below, we will present several typical examples of data manipulation. But first let's start with a quick example on how to access the pixel data.

```
>>> ad = astrodata.open('../playdata/N20170609S0154.fits')

>>> the_data = ad[1].data
>>> type(the_data)
<class 'numpy.ndarray'>

>>> # Loop through the extensions
>>> for ext in ad:
...     the_data = ext.data
...     print(the_data.sum())
...
```

In this example, we first access the pixels for the second extensions. Remember that in Python, list are zero-indexed, hence we access the second extension as `ad[1]`. The `.data` attribute contains a NumPy `ndarray`. In the for-loop, for each extension, we get the data and use the NumPy `.sum()` method to sum the pixel values. Anything that can be done with a `ndarray` can be done on `AstroData` pixel data.

6.2 Arithmetic on AstroData Objects

`AstroData` objects support basic in-place arithmetics with these methods:

addition	.add()
subtraction	.subtract()
multiplication	.multiply()
division	.divide()

Normal, not in-place, arithmetics is also possible using the standard operators, `+`, `-`, `*`, and `/`.

The big advantage of using `AstroData` to do arithmetics is that the variance and mask, if present, will be propagated through to the output `AstroData` object. We will explore the variance propagation in the next section and mask usage later in this chapter.

6.2.1 Simple operations

Here are a few examples of arithmetics on `AstroData` objects.:

```
>>> ad = astrodata.open('../playdata/N20170521S0925_forStack.fits')

>>> # Addition
>>> ad.add(50.)
>>> ad = ad + 50.
>>> ad += 50.

>>> # Subtraction
>>> ad.subtract(50.)
>>> ad = ad - 50.
>>> ad -= 50.

>>> # Multiplication (Using a descriptor)
>>> ad.multiply(ad.exposure_time())
>>> ad = ad * ad.exposure_time()
>>> ad *= ad.exposure_time()

>>> # Division (Using a descriptor)
>>> ad.divide(ad.exposure_time())
>>> ad = ad / ad.exposure_time()
>>> ad /= ad.exposure_time()
```

When the syntax `adout = adin + 1` is used, the output variable is a copy of the original. In the examples above we reassign the result back onto the original. The two other forms, `ad.add()` and `ad +=` are in-place operations.

When a descriptor returns a list because the value changes for each extension, a for-loop is needed:

```
>>> for (ext, gain) in zip(ad, ad.gain()):
...     ext.multiply(gain)
...
```

If you want to do the above but on a new object, leaving the original unchanged, use `deepcopy` first.

```
>>> from copy import deepcopy
>>> adcopy = deepcopy(ad)
>>> for (ext, gain) in zip(adcopy, adcopy.gain()):
...     ext.multiply(gain)
...
```

6.2.2 Operator Precedence

The `AstroData` arithmetics methods can be stringed together but beware that there is no operator precedence when that is done. For arithmetics that involve more than one operation, it is probably safer to use the normal Python operator syntax. Here is a little example to illustrate the difference.

```
>>> ad.add(5).multiply(10).subtract(5)

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

This is because the methods modify the object in-place, one operation after the other from left to right. This also means that the original is modified.

This example applies the expected operator precedence:

```
>>> ad = ad + ad * 3 - 40.
>>> # means: ad = ad + (ad * 3) - 40.
```

If you need a copy, leaving the original untouched, which is sometimes useful you can use `deepcopy` or just use the normal operator and assign to a new variable.:

```
>>> adnew = ad + ad * 3 - 40.
```

6.3 Variance

When doing arithmetic on an `AstroData` object, if a variance is present it will be propagated appropriately to the output no matter which syntax you use (the methods or the Python operators).

6.3.1 Adding a Variance Plane

In this example, we will add the poisson noise to an `AstroData` dataset. The data is still in ADU, therefore the poisson noise as variance is `signal / gain`. We want to set the variance for each of the pixel extensions.

```
>>> ad = astrodata.open('../playdata/N20170609S0154.fits')

>>> for (extension, gain) in zip(ad, ad.gain()):
...    extension.variance = extension.data / gain
...
```

Check `ad.info()`, you will see a variance plane for each of the four extensions.

6.3.2 Automatic Variance Propagation

As mentioned before, if present, the variance plane will be propagated to the resulting `AstroData` object when doing arithmetics. The variance calculation assumes that the data are not correlated.

Let's look into an example.

```
#     output = x * x
# var_output = var * x^2 + var * x^2
>>> ad = astrodata.open('../playdata/N20170609S0154_varAdded.fits')

>>> ad[1].data[50,50]
56.160931
>>> ad[1].variance[50,50]
96.356529
>>> adout = ad * ad
>>> adout[1].data[50,50]
3154.05
>>> adout[1].variance[50,50]
607826.62
```

6.4 Data Quality Plane

The NDData `mask` stores the data quality plane. The simplest form is a True/False array of the same size at the pixel array. In Astrodata we favor a bit array that allows for additional information about why the pixel is being masked. For example at Gemini here is our bit mapping for bad pixels.

Meaning	Value
Bad pixel	1
Non Linear	2
Saturated	4
Cosmic Ray	8
No Data	16
Overlap	32
Unilluminated	64

(These definitions are located in `geminidr.gemini.lookups.DQ_definitions`.)

So a pixel marked 10 in the mask, would be a "non-linear" "cosmic ray". The `AstroData` masks are propagated with bitwise-OR operation. For example, let's say that we are stacking frames. A pixel is set as bad (value 1) in one frame, saturated in another (value 4), and fine in all the other the frames (value 0). The mask of the resulting stack will be assigned a value of 5 for that pixel.

These bitmasks will work like any other NumPy True/False mask. There is a usage example below using the mask.

The mask can be accessed as follow:

```
>>> ad = astrodata.open('../playdata/N20170609S0154_varAdded.fits')
>>> ad.info()

>>> ad[2].mask
```

6.5 Display

Since the data is stored in the `AstroData` object as a NumPy `ndarray` any tool that works on `ndarray` can be used. To display to DS9 there is the `imexam` package. The `numdisplay` package is still available for now but it is no longer supported by STScI. We will show how to use `imexam` to display and read the cursor position. Read the documentation on that tool to learn more about what else it has to offer.

6.5.1 Displaying with imexam

Here is an example how to display pixel data to DS9 with `imexam`.

```
>>> import imexam
>>> ad = astrodata.open('../playdata/N20170521S0925_forStack.fits')

# Connect to the DS9 window (should already be opened.)
>>> ds9 = imexam.connect(list(imexam.list_active_ds9())[0])

>>> ds9.view(ad[0].data)

# To scale "a la IRAF"
>>> ds9.view(ad[0].data)
>>> ds9.scale('zscale')

# To set the mininum and maximum scale values
>>> ds9.view(ad[0].data)
>>> ds9.scale('limits 0 2000')
```

6.5.2 Retrieving cursor position with imexam

The function `readcursor()` can be used to retrieve cursor position in pixel coordinates. 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 tuple with three values will be returned: the x and y coordinates **in 0-based system**, and the value of the key the user hit.

```
>>> import imexam
>>> ad = astrodata.open('../playdata/N20170521S0925_forStack.fits')

# Connect to the DS9 window (should already be opened.)
# and display
>>> ds9 = imexam.connect(list(imexam.list_active_ds9())[0])
>>> ds9.view(ad[0].data)
>>> ds9.scale('zscale')


>>> cursor_coo = ds9.readcursor()
>>> print(cursor_coo)

# To extract only the x,y coordinates
>>> (xcoo, ycoo) = cursor_coo[:2]
>>> print(xcoo, ycoo)

# If you are also interested in the keystroke
```

(continues on next page)

(continued from previous page)

```
>>> keystroke = cursor_coo[2]
>>> print('You pressed this key: %s' % keystroke)
```

6.6 Useful tools from the NumPy, SciPy, and Astropy Packages

Like for the Display section, this section is not really specific to Astrodata but is rather a quick show-and-tell of a few things that can be done on the pixels with the big scientific packages NumPy, SciPy, and Astropy.

Those three packages are very large and rich. They have their own extensive documentation and it is highly recommend for the users to learn about what they have to offer. It might save you from re-inventing the wheel.

The pixels, the variance, and the mask are stored as NumPy `ndarray`'s. Let us go through some basic examples, just to get a feel for how the data in an `AstroData` object can be manipulated.

6.6.1 ndarray

The data are contained in NumPy `ndarray` objects. Any tools that works on an `ndarray` can be used with Astrodata.

```
>>> ad = astrodata.open('../playdata/N20170609S0154.fits')

>>> data = ad[0].data

>>> # Shape of the array.  (equivalent to NAXIS2, NAXIS1)
>>> data.shape
(2112, 288)

>>> # Value of a pixel at "IRAF" or DS9 coordinates (100, 50)
>>> data[49,99]
455

>>> # Data type
>>> data.dtype
dtype('uint16')
```

The two most important thing to remember for users coming from the IRAF world or the Fortran world are that the array has the y-axis in the first index, the x-axis in the second, and that the array indices are zero-indexed, not one-indexed. The examples above illustrate those two critical differences.

It is sometimes useful to know the data type of the values stored in the array. Here, the file is a raw dataset, fresh off the telescope. No operations has been done on the pixels yet. The data type of Gemini raw datasets is always "Unsigned integer (0 to 65535)", `uint16`.

Warning: Beware that doing arithmetic on `uint16` can lead to unexpected results. This is a NumPy behavior. If the result of an operation is higher than the range allowed by `uint16`, the output value will be "wrong". The data type will not be modified to accommodate the large value. A workaround, and a safety net, is to multiply the array by `1.0` to force the conversion to a `float64`.

```
>>> a = np.array([65535], dtype='uint16')
>>> a + a
array([65534], dtype=uint16)
>>> 1.0*a + a
array([ 131070.])
```

6.6.2 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.

```
>>> import numpy as np

>>> ad = astrodata.open('../playdata/N20170609S0154.fits')
>>> data = ad[0].data

>>> 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.

6.6.3 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 examples below, we calculated the clipped average of the first pixel extension with a rejection threshold set to +/- 3 times the standard deviation.

Before Astropy, it was possible to do something like that with only NumPy tools, like in this example:

```
>>> import numpy as np

>>> ad = astrodata.open('../playdata/N20170609S0154.fits')
>>> data = ad[0].data

>>> stddev = data.std()
>>> mean = data.mean()

>>> clipped_mean = np.ma.masked_outside(data, mean-3*stddev, mean+3*stddev).mean()
```

There is no iteration in that example. It is a straight one-time clipping.

For something more robust, there is an Astropy function that can help, in particular by adding an iterative process to the calculation. Here is how it is done:

```
>>> import numpy as np
>>> from astropy.stats import sigma_clip

>>> ad = astrodata.open('../playdata/N20170609S0154.fits')
>>> data = ad[0].data

>>> clipped_mean = np.ma.mean(sigma_clip(data, sigma=3))
```

6.6.4 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 functions for image processing. See the SciPy documentation for more information.

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

```
>>> from scipy.ndimage import filters
>>> import imexam

>>> ad = astrodata.open('../playdata/N20170521S0925_forStack.fits')
>>> data = ad[0].data

>>> # We need to prepare an array of the same size and shape as
>>> # the data array.  The result will be put in there.
>>> convolved_data = np.zeros(data.size).reshape(data.shape)

>>> # We now apply the convolution filter.
>>> sigma = 10.
>>> filters.gaussian_filter(data, sigma, output=convolved_data)

>>> # Let's visually compare the convolved image with the original
>>> ds9 = imexam.connect(list(imexam.list_active_ds9())[0])
>>> ds9.view(data)
>>> ds9.scale('zscale')
>>> ds9.frame(2)
>>> ds9.view(convolved_data)
>>> ds9.scale('zscale')
>>> ds9.blink()
>>> # When you are convinced it's been convolved, stop the blinking.
>>> ds9.blink(blink=False)
```

Note that there is an Astropy way to do this convolution, with tools in `astropy.convolution` package. Beware that for this particular kernel we have found that the Astropy `convolve` function is extremely slow compared to the SciPy solution. This is because the SciPy function is optimized for a Gaussian convolution while the generic `convolve` function in Astropy can take in any kernel. Being able to take in any kernel is a very powerful feature, but the cost is time. The lesson here is do your research, and find the best tool for your needs.

6.6.5 Many other tools

There many, many other tools available out there. Here are the links to the three big projects we have featured in this section.

- NumPy: www.numpy.org
- SciPy: www.scipy.org
- Astropy: www.astropy.org

6.7 Using the Astrodata Data Quality Plane

Let us look at an example where the use of the Astrodata mask is necessary to get correct statistics. A GMOS imaging frame has large sections of unilluminated pixels; the edges are not illuminated and there are two bands between the three CCDs that represent the physical gap between the CCDs. Let us have a look at the pixels to have a better sense of the data:

```
>>> ad = astrodata.open('../playdata/N20170521S0925_forStack.fits')
>>> import imexam
>>> ds9 = imexam.connect(list(imexam.list_active_ds9())[0])
```

(continues on next page)

(continued from previous page)

```
>>> ds9.view(ad[0].data)
>>> ds9.scale('zscale')
```

See how the right and left portions of the frame are not exposed to the sky, and the 45 degree angle cuts of the four corners. The chip gaps too. If we wanted to do statistics on the whole frames, we certainly would not want to include those unilluminated areas. We would want to mask them out.

Let us have a look at the mask associated with that image:

```
>>> ds9.view(ad[0].mask)
>>> ds9.scale('zscale')
```

The bad sections are all white (pixel value > 0). There are even some illuminated pixels that have been marked as bad for a reason or another.

Let us use that mask to reject the pixels with no or bad information and do calculations only on the good pixels. For the sake of simplicity we will just do an average. This is just illustrative. We show various ways to accomplish the task; choose the one that best suits your need or that you find most readable.

```
>>> import numpy as np

>>> # For clarity...
>>> data = ad[0].data
>>> mask = ad[0].mask

>>> # Reject all flagged pixels and calculate the mean
>>> np.mean(data[mask == 0])
>>> np.ma.masked_array(data, mask).mean()

>>> # Reject only the pixels flagged "no_data" (bit 16)
>>> np.mean(data[(mask & 16) == 0])
>>> np.ma.masked_array(data, mask & 16).mean()
>>> np.ma.masked_where(mask & 16, data).mean()
```

The “long” form with `np.ma.masked_*` is useful if you are planning to do more than one operation on the masked array. For example:

```
>>> clean_data = np.ma.masked_array(data, mask)
>>> clean_data.mean()
>>> np.ma.median(clean_data)
>>> clean_data.max()
```

6.8 Manipulate Data Sections

So far we have shown examples using the entire data array. It is possible to work on sections of that array. If you are already familiar with Python, you probably already know how to do most if not all of what is in this section. For readers new to Python, and especially those coming from IRAF, there are a few things that are worth explaining.

When indexing a NumPy `ndarray`, the left most number refers to the highest dimension’s axis. For example, in a 2D array, the IRAF section are in (x-axis, y-axis) format, while in Python they are in (y-axis, x-axis) format. Also important to remember is that the `ndarray` is 0-indexed, rather than 1-indexed like in Fortran or IRAF.

Putting it all together, a pixel position (x,y) = (50,75) in IRAF or from the cursor on a DS9 frame, is accessed in Python as `data[74,49]`. Similarly, the IRAF section [10:20, 30:40] translate in Python to [9:20, 29:40]. Also remember that when slicing in Python, the upper limit of the slice is not included in the slice. This is why here we request 20 and 40 rather 19 and 39.

Let's put it in action.

6.8.1 Basic Statistics on Section

In this example, we do simple statistics on a section of the image.

```
>>> import numpy as np

>>> ad = astrodata.open('../playdata/N20170521S0925_forStack.fits')
>>> data = ad[0].data

>>> # Get statistics for a 25x25 pixel-wide box centered on pixel
>>> # (50,75)  (DS9 frame coordinate)
>>> xc = 49
>>> yc = 74
>>> buffer = 25
>>> (xlow, xhigh) = (xc - buffer//2, xc + buffer//2 + 1)
>>> (ylow, yhigh) = (yc - buffer//2, yc + buffer//2 + 1)
>>> # The section is [62:87, 37:62]
>>> stamp = data[ylow:yhigh, xlow:xhigh]
>>> mean = stamp.mean()
>>> median = np.median(stamp)
>>> stddev = stamp.std()
>>> minimum = stamp.min()
>>> maximum = stamp.max()

>>> print(' Mean   Median  Stddev  Min   Max\n \
... %.2f  %.2f   %.2f    %.2f  %.2f' % \
... (mean, median, stddev, minimum, maximum))
```

Have you noticed that the median is calculated with a function rather than a method? This is simply because the `ndarray` object does not have a method to calculate the median.

6.8.2 Example - Overscan Subtraction with Trimming

Several concepts from previous sections and chapters are used in this example. The Descriptors are used to retrieve the overscan section and the data section information from the headers. Statistics are done on the NumPy `ndarray` representing the pixel data. Astrodata arithmetics is used to subtract the overscan level. Finally, the overscan section is trimmed off and the modified `AstroData` object is written to a new file on disk.

To make the example more complete, and to show that when the pixel data array is trimmed, the variance (and mask) arrays are also trimmed, let us add a variance plane to our raw data frame.

```
>>> ad = astrodata.open('../playdata/N20170609S0154.fits')

>>> for (extension, gain) in zip(ad, ad.gain()):
...    extension.variance = extension.data / gain
...

>>> # Here is how the data structure looks like before the trimming.
>>> ad.info()
Filename: ../playdata/N20170609S0154.fits
Tags: ACQUISITION GEMINI GMOS IMAGE NORTH RAW SIDEREAL UNPREPARED
```

(continues on next page)

(continued from previous page)

```
Pixels Extensions
Index  Content                  Type              Dimensions     Format
[ 0]   science                  NDAstroData       (2112, 288)    uint16
          .variance             ndarray           (2112, 288)    float64
[ 1]   science                  NDAstroData       (2112, 288)    uint16
          .variance             ndarray           (2112, 288)    float64
[ 2]   science                  NDAstroData       (2112, 288)    uint16
          .variance             ndarray           (2112, 288)    float64
[ 3]   science                  NDAstroData       (2112, 288)    uint16
          .variance             ndarray           (2112, 288)    float64

>>> # Let's operate on the first extension.
>>> #
>>> # The section descriptors return the section in a Python format
>>> # ready to use, 0-indexed.
>>> oversec = ad[0].overscan_section()
>>> datasec = ad[0].data_section()

>>> # Measure the overscan level
>>> mean_overscan = ad[0].data[oversec.y1: oversec.y2, oversec.x1: oversec.x2].mean()

>>> # Subtract the overscan level.  The variance will be propagated.
>>> ad[0].subtract(mean_overscan)

>>> # Trim the data to remove the overscan section and keep only
>>> # the data section.
>>> #
>>> # Here we work on the NDAstroData object to have the variance
>>> # trimmed automatically to the same size as the science array.
>>> # To reassign the cropped NDAstroData, we use the reset() method.
>>> ad[0].reset(ad[0].nddata[datasec.y1:datasec.y2, datasec.x1:datasec.x2])

>>> # Now look at the dimensions of the first extension, science
>>> # and variance.  That extension is smaller than the others.
>>> ad.info()
Filename: ../playdata/N20170609S0154.fits
Tags: ACQUISITION GEMINI GMOS IMAGE NORTH RAW SIDEREAL UNPREPARED

Pixels Extensions
Index  Content                  Type              Dimensions     Format
[ 0]   science                  NDAstroData       (2112, 256)    float64
          .variance             ndarray           (2112, 256)    float64
[ 1]   science                  NDAstroData       (2112, 288)    uint16
          .variance             ndarray           (2112, 288)    float64
[ 2]   science                  NDAstroData       (2112, 288)    uint16
          .variance             ndarray           (2112, 288)    float64
[ 3]   science                  NDAstroData       (2112, 288)    uint16
          .variance             ndarray           (2112, 288)    float64

>>> # We can write this to a new file
>>> ad.write('partly_overscan_corrected.fits')
```

A new feature presented in this example is the ability to work on the `NDAstroData` object directly. This is particularly useful when cropping the science pixel array as one will want the variance and the mask arrays cropped exactly the same way. Taking a section of the `NDAstroData` object (ad[0].nddata[y1:y2, x1:x2]), instead of just the *.data* array, does all that for us.

To reassign the cropped `NDAstroData` to the extension one uses the `.reset()` method as shown in the example.

Of course to do the overscan correction correctly and completely, one would loop over all four extensions. But that's the only difference.

6.9 Data Cubes

Reduced Integral Field Unit (IFU) data is commonly represented as a cube, a three-dimensional array. The `data` component of an `AstroData` object extension can be such a cube, and it can be manipulated and explored with NumPy, AstroPy, SciPy, imexam, like we did already in this section with 2D arrays. We can use matplotlib to plot the 1D spectra represented in the third dimension.

In Gemini IFU cubes, the first axis is the X-axis, the second, the Y-axis, and the wavelength is in the third axis. Remember that in a `ndarray` that order is reversed (wlen, y, x).

In the example below we "collapse" the cube along the wavelenth axis to create a "white light" image and display it. Then we plot a 1D spectrum from a given (x,y) position.

```
>>> import imexam
>>> import matplotlib.pyplot as plt

>>> ds9 = imexam.connect(list(imexam.list_active_ds9())[0])

>>> adcube = astrodata.open('../playdata/gmosifu_cube.fits')
>>> adcube.info()

>>> # Sum along the wavelength axis to create a "white light" image
>>> summed_image = adcube[0].data.sum(axis=0)
>>> ds9.view(summed_image)
>>> ds9.scale('minmax')

>>> # Plot a 1-D spectrum from the spatial position (14,25).
>>> plt.plot(adcube[0].data[:,24,13])
>>> plt.show()   # might be needed, depends on matplotlibrc interactive setting
```

Now that is nice but it would be nicer if we could plot the x-axis in units of Angstroms instead of pixels. We use the AstroData's WCS handler, which is based on `gwcs.wcs.WCS` to get the necessary information. A particularity of `gwcs.wcs.WCS` is that it refers to the axes in the "natural" way, (x, y, wlen) contrary to Python's (wlen, y, x). It truly requires you to pay attention.

```
>>> import matplotlib.pyplot as plt

>>> adcube = astrodata.open('../playdata/gmosifu_cube.fits')

# We get the wavelength axis in Angstroms at the position we want to
# extract, x=13, y=24.
# The wcs call returns a 3-element list, the third element ([2]) contains
# the wavelength values for each pixel along the wavelength axis.

>>> length_wlen_axis = adcube[0].shape[0]    # (wlen, y, x)
>>> wavelengths = adcube[0].wcs(13, 24, range(length_wlen_axis))[2] # (x, y, wlen)

# We get the intensity along that axis
>>> intensity = adcube[0].data[:, 24, 13]    # (wlen, y, x)

# We plot
```

(continues on next page)

(continued from previous page)

```
plt.clf()
plt.plot(wavelengths, intensity)
plt.show()
```

6.10 Plot Data

The main plotting package in Python is `matplotlib`. We have used it in the previous section on data cubes to plot a spectrum. There is also relatively new project called `imexam` which provides astronomy-specific tools for the exploration and measurement of data. We have also used that package above to display images to DS9.

In this section we absolutely do not aim at covering all the features of either package but rather to give a few examples that can get the readers started in their exploration of the data and of the visualization packages.

Refer to the projects web pages for full documentation.

- Matplotlib: https://matplotlib.org
- imexam: https://github.com/spacetelescope/imexam

6.10.1 Matplotlib

With Matplotlib you have full control on your plot. You do have to do a bit for work to get it perfect though. However it can produce publication quality plots. Here we just scratch the surface of Matplotlib.

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> from astropy import wcs

>>> ad_image = astrodata.open('../playdata/N20170521S0925_forStack.fits')
>>> ad_spectrum = astrodata.open('../playdata/estgsS20080220S0078.fits')

>>> # Line plot from image.  Row #1044 (y-coordinate)
>>> line_index = 1043
>>> line = ad_image[0].data[line_index, :]
>>> plt.clf()
>>> plt.plot(line)
>>> plt.show()

>>> # Column plot from image, averaging across 11 pixels around colum #327
>>> col_index = 326
>>> width = 5
>>> xlow = col_index - width
>>> xhigh = col_index + width + 1
>>> thick_column = ad_image[0].data[:, xlow:xhigh]
>>> plt.clf()
>>> plt.plot(thick_column.mean(axis=1))  # mean along the width.
>>> plt.show()
>>> plt.ylim(0, 50)     # Set the y-axis range
>>> plt.plot(thick_column.mean(axis=1))
>>> plt.show()

>>> # Contour plot for a section of an image.
>>> center = (1646, 2355)
>>> width = 15
```

(continues on next page)

(continued from previous page)

```
>>> xrange = (center[1]-width//2, center[1] + width//2 + 1)
>>> yrange = (center[0]-width//2, center[0] + width//2 + 1)
>>> blob = ad_image[0].data[yrange[0]:yrange[1], xrange[0]:xrange[1]]
>>> plt.clf()
>>> plt.imshow(blob, cmap='gray', origin='lower')
>>> plt.contour(blob)
>>> plt.show()

>>> # Spectrum in pixels
>>> plt.clf()
>>> plt.plot(ad_spectrum[0].data)
>>> plt.show()

>>> # Spectrum in Angstroms
>>> spec_wcs = wcs.WCS(ad_spectrum[0].hdr)
>>> pixcoords = np.array(range(ad_spectrum[0].data.shape[0]))
>>> wlen = spec_wcs.wcs_pix2world(pixcoords, 0)[0]
>>> plt.clf()
>>> plt.plot(wlen, ad_spectrum[0].data)
>>> plt.show()
```

6.10.2 imexam

For those who have used IRAF, `imexam` is a well-known tool. The Python `imexam` reproduces many of its IRAF predecesor, the interactive mode of course, but it also offers programmatic tools. One can even control DS9 from Python. As for Matplotlib, here we really just scratch the surface of what `imexam` has to offer.

```
 >>> import imexam
 >>> from imexam.imexamine import Imexamine

 >>> ad_image = astrodata.open('../playdata/N20170521S0925_forStack.fits')

 # Display the image
 >>> ds9 = imexam.connect(list(imexam.list_active_ds9())[0])
 >>> ds9.view(ad_image[0].data)
 >>> ds9.scale('zscale')

 # Run in interactive mode.  Try the various commands.
>>> ds9.imexam()

 # Use the programmatic interface
 # First initialize an Imexamine object.
 >>> plot = Imexamine()

 # Line plot from image.  Row #1044 (y-coordinate)
 >>> line_index = 1043
 >>> plot.plot_line(0, line_index, ad_image[0].data)

 # Column plot from image, averaging across 11 pixels around colum #327
 # There is no setting for this, so we have to do something similar
 # to what we did with matplotlib.
 >>> col_index = 326
 >>> width = 5
 >>> xlow = col_index - width
 >>> xhigh = col_index + width + 1
```

(continues on next page)

(continued from previous page)

```
>>> thick_column = ad_image[0].data[:, xlow:xhigh]
>>> mean_column = thick_column.mean(axis=1)
>>> plot.plot_column(0, 0, np.expand_dims(mean_column, 1))

>>> # Contour plot for a section of an image.
>>> center = (1646, 2355)  # in python coordinates
>>> width = 15
>>> plot.contour_pars['ncolumns'][0] = width
>>> plot.contour_pars['nlines'][0] = width
>>> plot.contour(center[1], center[0], ad_image[0].data)
```

CHAPTER 7

Table Data

Try it yourself

Download the data package (*Try it yourself*) if you wish to follow along and run the examples. Then

```
$ cd <path>/ad_usermanual/playground
$ python
```

Then import core astrodata and the Gemini astrodata configurations.

```
>>> import astrodata
>>> import gemini_instruments
```

7.1 Tables and Astrodata

Tables are stored as `astropy.table Table` class. FITS tables too are represented in Astrodata as `Table` and FITS headers are stored in the NDAstroData *.meta* attribute. Most table access should be done through the `Table` interface. The best reference on `Table` is the Astropy documentation itself. In this chapter we covers some common examples to get the reader started.

The `astropy.table` documentation can be found at: http://docs.astropy.org/en/stable/table/index.html

7.2 Operate on a Table

Let us open a file with tables. Some tables are associated with specific extensions, and there is one table that is global to the *AstroData* object.

```
>>> ad = astrodata.open('../playdata/N20170609S0154_varAdded.fits')
>>> ad.info()
```

To access the global table named `REFCAT`:

```
>>> ad.REFCAT
```

To access the OBJCAT table in the first extension

```
>>> ad[0].OBJCAT
```

7.2.1 Column and Row Operations

Columns are named. Those names are used to access the data as columns. Rows are not names and are simply represented as a sequential list.

Read columns and rows

To get the names of the columns present in the table:

```
>>> ad.REFCAT.colnames
['Id', 'Cat_Id', 'RAJ2000', 'DEJ2000', 'umag', 'umag_err', 'gmag',
'gmag_err', 'rmag', 'rmag_err', 'imag', 'imag_err', 'zmag', 'zmag_err',
'filtermag', 'filtermag_err']
```

Then it is easy to request the values for specific columns:

```
>>> ad.REFCAT['zmag']
>>> ad.REFCAT['zmag', 'zmag_err']
```

To get the content of a specific row, row 10 in this case:

```
>>> ad.REFCAT[9]
```

To get the content of a specific row(s) from a specific column(s):

```
>>> ad.REFCAT['zmag'][4]
>>> ad.REFCAT['zmag'][4:10]
>>> ad.REFCAT['zmag', 'zmag_err'][4:10]
```

Change values

Assigning new values works in a similar way. When working on multiple elements it is important to feed a list that matches in size with the number of elements to replace.

```
>>> ad.REFCAT['imag'][4] = 20.999
>>> ad.REFCAT['imag'][4:10] = [5, 6, 7, 8, 9, 10]

>>> overwrite_col = [0] * len(ad.REFCAT)  # a list of zeros, size = nb of rows
>>> ad.REFCAT['imag_err'] = overwrite_col
```

Add a row

To append a row, there is the add_row() method. The length of the row should match the number of columns:

```
>>> new_row = [0] * len(ad.REFCAT.colnames)
>>> new_row[1] = ''   # Cat_Id column is of "str" type.
>>> ad.REFCAT.add_row(new_row)
```

Add a column

Adding a new column can be more involved. If you need full control, please see the AstroPy Table documentation. For a quick addition, which might be sufficient for your use case, we simply use the "dictionary" technique. Please note that when adding a column, it is important to ensure that all the elements are of the same type. Also, if you are planning to use that table in IRAF/PyRAF, we recommend not using 64-bit types.

```
>>> import numpy as np

>>> new_column = [0] * len(ad.REFCAT)
>>> # Ensure that the type is int32, otherwise it will default to int64
>>> # which generally not necessary.  Also, IRAF 32-bit does not like it.
>>> new_column = np.array(new_column).astype(np.int32)
>>> ad.REFCAT['my_column'] = new_column
```

If you are going to write that table back to disk as a FITS Bintable, then some additional headers need to be set. Astrodata will take care of that under the hood when the *write* method is invoked.

```
>>> ad.write('myfile_with_modified_table.fits')
```

7.2.2 Selection and Rejection Operations

Normally, one does not know exactly where the information needed is located in a table. Rather some sort of selection needs to be done. This can also be combined with various calculations. We show two such examples here.

Select a table element from criterion

```
>>> # Get the magnitude of a star selected by ID number
>>> ad.REFCAT['zmag'][ad.REFCAT['Cat_Id'] == '1237662500002005475']

>>> # Get the ID and magnitude of all the stars brighter than zmag 18.
>>> ad.REFCAT['Cat_Id', 'zmag'][ad.REFCAT['zmag'] < 18.]
```

Rejection and selection before statistics

```
>>> t = ad.REFCAT   # to save typing

>>> # The table has "NaN" values.  ("Not a number")  We need to ignore them.
>>> t['zmag'].mean()
nan
>>> # applying rejection of NaN values:
>>> t['zmag'][np.where(~np.isnan(t['zmag']))].mean()
20.377306
```

7.2.3 Accessing FITS table headers directly

If for some reason you need to access the FITS table headers directly, here is how to do it. It is very unlikely that you will need this.

To see the FITS headers:

```
>>> ad.REFCAT.meta['header']
>>> ad[0].OBJCAT.meta['header']
```

To retrieve a specific FITS table header:

```
>>> ad.REFCAT.meta['header']['TTYPE3']
'RAJ2000'
>>> ad[0].OBJCAT.meta['header']['TTYPE3']
'Y_IMAGE'
```

To retrieve all the keyword names matching a selection:

```
>>> keynames = [key for key in ad.REFCAT.meta['header'] if key.startswith('TTYPE')]
```

7.3 Create a Table

To create a table that can be added to an `AstroData` object and eventually written to disk as a FITS file, the first step is to create an Astropy `Table`.

Let us first add our data to NumPy arrays, one array per column:

```
>>> import numpy as np

>>> snr_id = np.array(['S001', 'S002', 'S003'])
>>> feii = np.array([780., 78., 179.])
>>> pabeta = np.array([740., 307., 220.])
>>> ratio = pabeta / feii
```

Then build the table from that data:

```
>>> from astropy.table import Table

>>> my_astropy_table = Table([snr_id, feii, pabeta, ratio],
...                          names=('SNR_ID', 'FeII', 'PaBeta', 'ratio'))
```

Now we append this Astropy `Table` to a new `AstroData` object.

```
>>> # Since we are going to write a FITS, we build the AstroData object
>>> # from FITS objects.
>>> from astropy.io import fits

>>> phu = fits.PrimaryHDU()
>>> ad = astrodata.create(phu)
>>> ad.append(my_astropy_table, name='MYTABLE')
>>> ad.info()
>>> ad.MYTABLE

>>> ad.write('new_table.fits')
```

APPENDIX A

List of Gemini Standard Descriptors

To run and re-use Gemini primitives and functions this list of Standard Descriptors must be defined for input data. This also applies to data that is to be served by the Gemini Observatory Archive (GOA).

For any `AstroData` objects, to get the list of the descriptors that are defined use the `AstroData.descriptors` attribute:

```
>>> import astrodata
>>> import gemini_instruments
>>> ad = astrodata.open('../playdata/N20170609S0154.fits')

>>> ad.descriptors
('airmass', 'amp_read_area', 'ao_seeing', ..., 'well_depth_setting')
```

To get the values:

```
>>> ad.airmass()

>>> for descriptor in ad.descriptors:
...     print(descriptor, getattr(ad, descriptor)())
```

Note that not all of the descriptors below are defined for all of the instruments. For example, `shuffle_pixels` is defined only for GMOS data since only GMOS offers a Nod & Shuffle mode.

Descriptor	**Short Definition**	**Python type**
		ad[0].desc()
		ad.desc()
airmass	Airmass of the observation.	float
amp_read_area	Combination of amplifier name and 1-indexed section relative to the detector.	str
		list of str
ao_seeing	Estimate of the natural seeing as calculated from the adaptive optics systems.	float
array_name	Name assigned to the array generated by a given amplifier, one array per amplifier.	str

Continued on next page

Table 1 – continued from previous page

Descriptor	Short Definition	Python type
		ad[0].desc()
		ad.desc()
		list of 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). Uses `namedtuple` "Section" defined in `gemini_instruments.common`.	Section
		list of Section
azimuth	Pointing position in azimuth, in degrees.	float
calibration_key	Key used in the database that the `getProcessed*` primitives use to store previous calibration association information.	str
camera	Name of the camera.	str
cass_rotator_pa	Position angle of the Cassegrain rotator, in degrees.	float
central_wavelength	Central wavelength, in meters.	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. Uses `namedtuple` "Section" defined in `gemini_instruments.common`	Section
		list of Section
dec	Declination of the center of the field, in degrees.	float
decker	Name of the decker.	str
detector_name	Name assigned to the detector.	str
detector_roi_setting	Human readable Region of Interest (ROI) setting	str
detector_rois_requested	Section defining the Regions of Interest, in 0-indexed pixels. Uses `namedtuple` "Section" defined in `gemini_instruments.common`.	list of Section
detector_section	Section covered by the detector(s), in 0-indexed pixels, relative to the whole mosaic of detectors. Uses `namedtuple` "Section" defined in `gemini_instruments.common`.	list
		list of Section
detector_x_bin	X-axis binning.	int
detector_x_offset	Telescope offset along the detector X-axis, in pixels.	float
detector_y_bin	Y-axis binning.	int
detector_y_offset	Telescope offset along the detector Y-axis, in pixels.	float
disperser	Name of the disperser.	str
dispersion	Value for the dispersion, in meters per pixel.	float
		list of float
dispersion_axis	Dispersion axis.	int
		list of int
effective_wavelength	Wavelength representing the bandpass or the spectrum coverage.	float
elevation	Pointing position in elevation, in degrees.	float
exposure_time	Exposure time, in seconds.	float
filter_name	Name of the filter combination.	str
focal_plane_mask	Name of the mask in the focal plane.	str
gain	Gain in electrons per ADU	float
		list of float
gain_setting	Human readable gain setting (eg. low, high)	str
gcal_lamp	Returns the name of the GCAL lamp being used, or "Off" if no lamp is in use.	str

Continued on next page

Table 1 – continued from previous page

Descriptor	Short Definition	Python type
		ad[0].desc()
		ad.desc()
group_id	Gemini observation group ID that identifies compatible data.	str
instrument	Name of the instrument	str
is_ao	Whether or not the adaptive optics system was used.	bool
is_coadds_summed	Whether co-adds are summed or averaged.	bool
local_time	Local time.	datetime
lyot_stop	Name of the lyot stop.	str
mdf_row_id	Mask Definition File row ID of a cut MOS or XD spectrum.	int ??
nod_count	Number of nods to A and B positions.	tuple of int
nod_offsets	Nod offsets to A and B positions, in arcseconds	tuple of float
nominal_atmospheric_extinction	Nomimal atmospheric extinction, from model.	float
nominal_photometric_zeropoint	Nominal photometric zeropoint.	float
		list of float
non_linear_level	Lower boundary of the non-linear regime.	float
		list of int
object	Name of the target (as entered by the user).	str
observation_class	Gemini class name for the observation (eg. 'science', 'acq', 'dayCal').	str
observation_epoch	Observation epoch.	float
observation_id	Gemini observation ID.	str
observation_type	Gemini observation type (eg. 'OBJECT', 'FLAT', 'ARC').	str
overscan_section	Section where the overscan data falls, in 0-indexed pixels. Uses namedtuple "Section" defined in `gemini_instruments.common`.	Section
		list of Section
pixel_scale	Pixel scale in arcsec per pixel.	float
program_id	Gemini program ID.	str
pupil_mask	Name of the pupil mask.	str ??
qa_state	Gemini 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
		list of 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
		list of int
shuffle_pixels	Charge shuffle, in pixels. (nod and shuffle mode)	int

Continued on next page

Table 1 – continued from previous page

Descriptor	Short Definition	Python type
		ad[0].desc()
		ad.desc()
slit	Name of the slit.	str
target_dec	Declination of the target, in degrees.	float
target_ra	Right Ascension of the target, in degrees.	float
telescope	Name of the telescope.	str
telescope_x_offset	Offset along the telescope's x-axis.	float
telescope_y_offset	Offset along the telescope's y-axis.	float
ut_date	UT date of the observation.	datetime.date
ut_datetime	UT date and time of the observation.	datetime
ut_time	UT time of the observation.	datetime.time
wavefront_sensor	Wavefront sensor used for the observation.	str
wavelength_band	Band associated with the filter or the central wavelength.	str
wcs_dec	Declination of the center of field from the WCS keywords. In degrees.	float
wcs_ra	Right Ascension of the center of field from the WCS keywords. In degrees.	float
well_depth_setting	Human readable well depth setting (eg. shallow, deep)	str