A User's Guide to Descriptors

Release 1.0.0

Emma Hogan

March 04, 2016

CONTENTS

CHAPTER

ONE

WHAT ARE DESCRIPTORS?

Descriptors are designed such that essential keyword values that describe a particular concept can be accessed from the headers of any dataset in a consistent manner, regardless of which instrument was used to obtain the data. This is particularly useful for Gemini data, since the majority of keywords used to describe a particular concept at Gemini are not uniform between the instruments.

CHAPTER

TWO

BASIC DESCRIPTOR USAGE

The command `typewalk -l` lists all the descriptors that are defined. As of the date of this document there are 73 descriptors defined (*Appendix A*).

The following commands show an example of how to use descriptors. They can be entered at an interactive Python prompt (e.g., `python`, `ipython`, `pyraf`):

```
>>> from astrodata import AstroData
# Load the fits file into AstroData
>>> ad = AstroData("N20091027S0137.fits")
# Count the number of pixel data extensions in the AstroData object
>>> ad.count_exts()
3
# Count the number of science extensions in the AstroData object
>>> ad.count_exts(extname="SCI")
3
# Get the airmass value using the airmass descriptor
>>> airmass = ad.airmass()
>>> print airmass
1.327
# Get the instrument name using the instrument descriptor
>>> print "My instrument is %s" % ad.instrument()
My instrument is GMOS-N
# Get the gain value for each pixel data extension
>>> for ext in ad:
...     print ext.gain()
...
2.1
2.337
2.3
>>> print ad.gain()
{('SCI', 2): 2.3370000000000002, ('SCI', 1): 2.1000000000000001,
('SCI', 3): 2.2999999999999998}
```

In the examples above, the airmass and instrument apply to the dataset as a whole i.e., the keywords themselves exist only in the Primary Header Unit (PHU) and so only one value is returned. However, the gain applies specifically to the pixel data extensions within the dataset and so for this AstroData object, since there are three pixel data extensions, three values are returned in the form of a Python dictionary, where the key of the dictionary is the ("`EXTNAME`", `EXTVER`) tuple.

For those descriptors that describe a concept applying specifically to the pixel data extensions within a dataset i.e., those that access keywords in the headers of the pixel data extensions, a value for every pixel data extension in the AstroData object is returned by default:

```
>>> new_ad = AstroData("stgsS20100223S0042.fits")
# Count the number of pixel data extensions in the AstroData object
```

```
>>> new_ad.count_exts()
103
# Count the number of science extensions in the AstroData object
>>> new_ad.count_exts(extname="SCI")
34
# Get the data section value for each pixel data extension in the form of a
# dictionary
>>> print new_ad.data_section()
{('SCI', 29): [0, 3108, 0, 22], ('DQ', 5): [0, 3108, 0, 21],
('DQ', 9): [0, 3108, 0, 21], ('SCI', 15): [0, 3108, 0, 22],
('SCI', 24): [0, 3108, 0, 21], ('VAR', 8): [0, 3108, 0, 21],
...
```

Descriptors can also be used to return values relating to a subset of pixel data extensions, i.e., those associated with a particular `EXTNAME`:

```
>>> new_ad.count_exts(extname="DQ")
34
# Get the data section value for the data quality extensions only
>>> print new_ad["DQ"].data_section()
{('DQ', 11): [0, 3108, 0, 21], ('DQ', 6): [0, 3108, 0, 22],
('DQ', 29): [0, 3108, 0, 22], ('DQ', 9): [0, 3108, 0, 21],
('DQ', 4): [0, 3108, 0, 22], ('DQ', 19): [0, 3108, 0, 21],
...
```

Note that it is quicker to obtain the values of a descriptor in the form of a dictionary than it is to obtain the values of a descriptor for each extension separately.

CHAPTER

THREE

ADVANCED DESCRIPTOR USAGE

3.1 DescriptorValue

When a descriptor is called, what is actually returned is a `DescriptorValue` (`DV`) object:

```
>>> from astrodata import AstroData
# Load the fits file into AstroData
>>> ad = AstroData("N20091027S0137.fits")
# Get the airmass value using the airmass descriptor
>>> airmass = ad.airmass()
>>> print airmass
1.327
>>> print type(airmass)
<class 'astrodata.Descriptors.DescriptorValue'>
```

Each descriptor has a default Python type defined:

```
>>> print ad.airmass().pytype
<type 'float'>
```

If any of the following operations are applied to the `DV` object, the `DV` object is automatically cast to the default Python type for that descriptor:

```
+   -   *   /   //   %   **   <<   >>   ^   <   <=   >   >=   ==
```

For example:

```
>>> print type(airmass*1.0)
<type 'float'>
```

The value of the descriptor can be retrieved with the default Python type by using `as_pytype()`:

```
>>> elevation = ad.elevation().as_pytype()
>>> print elevation
48.6889222222
>>> print type(elevation)
<type 'float'>
```

The `as_pytype()` member function of the `DV` object should only be required when the `DV` object can not be automatically cast to it's default Python type. For example, when it is necessary to use the actual value of a descriptor (e.g., string, float, etc.) rather than a `DV` object as a key of a Python dictionary, the `DV` object can be cast to it's default Python type using `as_pytype()`:

```
>>> my_key = ad.gain_setting.as_pytype()
>>> my_value = my_dict[my_key]
```

If an alternative Python type is required for whatever reason, the DV object can be cast to the appropriate Python type as follows:

```
>>> elevation_as_int = int(ad.elevation())
>>> print elevation_as_int
48
>>> print type(elevation_as_int)
<type 'int'>
```

When using operations with a DV object and a numpy object, care must be taken. Consider the following cases:

```
>>> ad[0].data / ad.gain()
>>> ad[0].data / ad.gain().as_pytype()
>>> ad[0].data / 12.345
```

All the above commands return the same result (assuming that ad.gain() = 12.345). However, the first command is extremely slow while the second and third commands are fast. In the first case, since both operands have overloaded operators, the operator from the operand on the left will be used. For some reason, the __div__ operator from the numpy object loops over each pixel in the numpy object and uses the DV object as an argument, which is very time consuming. Therefore, the DV object should be cast to an appropriate Python type before using it in an operation with a numpy object.

In the case where a descriptor returns multiple values (one for each pixel data extension), a Python dictionary is used to store the values, where the key of the dictionary is the ("EXTNAME", EXTVER) tuple:

```
>>> print ad.gain()
{('SCI', 2): 2.3370000000000002, ('SCI', 1): 2.1000000000000001,
('SCI', 3): 2.2999999999999998}
```

For those descriptors that describe a concept applying specifically to the pixel data extensions within a dataset but has the same value for each pixel data extension will "act" as a single value that has a Python type defined by the default Python type of that descriptor:

```
>>> xbin = ad.detector_x_bin()
>>> print xbin
2
>>> print type(xbin)
<class 'astrodata.Descriptors.DescriptorValue'>
>>> print xbin.pytype
<type 'int'>
```

If the original value of the descriptor is required, it can be retrieved by using get_value():

```
>>> xbin = ad.detector_x_bin().get_value()
>>> print xbin
{('SCI', 2): 2, ('SCI', 1): 2, ('SCI', 3): 2}
>>> print type(xbin)
<type 'dict'>
>>> print xbin[("SCI", 1)]
2
```

3.2 DescriptorUnits

The DescriptorUnits (DU) object provides a way to access and update the units for a given descriptor. This feature is not yet implemented, but development is ongoing.

CHAPTER

FOUR

WRITING AND ADDING NEW DESCRIPTORS

4.1 Overview of the Descriptor Code

The infrastructure for the descriptor code is located within the `astrodata` subpackage in the `gemini_python` package. In addition, a default set of descriptors that access FITS standard keywords is provided by `astrodata`. This code is generic and is not Gemini specific.

The Gemini descriptor code is contained within the `astrodata_Gemini` addon in the `gemini_python` package. The Gemini descriptors inherit the FITS descriptors, since the Gemini telescopes produce files in FITS format. Other telescope / instrument addons can exist, enabling the addition of descriptors from non-Gemini instruments.

The following sections will refer specifically to adding descriptors to the already existing Gemini descriptor code contained within the `astrodata_Gemini` addon.

4.2 Overview of the FITS Descriptor Code

The FITS descriptors provide access to those keywords that are part of the FITS standard. There are currently 53 keywords defined in the FITS standard (http://heasarc.gsfc.nasa.gov/docs/fcg/standard_dict.html). There are four FITS descriptors available that return the value of the corresponding FITS standard keywords in the PHU of the AstroData object:

- `instrument [INSTRUME]`
- `object [OBJECT]`
- `telescope [TELESCOP]`
- `ut_date [DATE-OBS]`

The FITS descriptor code is currently located in the `gemini_python` package in the `astrodata_FITS/ADCONFIG_FITS/descriptors` directory [NOTE: the `astrodata_FITS` directory will be moved to the `astrodata` directory in the near future]:

- `calculatorIndex.FITS.py`
- `CalculatorInterface_FITS.py`
- `DescriptorsList_FITS.py`
- `docstrings.py`
- `FITS_Descriptors.py`
- `FITS_Keywords.py`

The function of each of these files is the same as the corresponding files in the Gemini descriptor code and will be described in the following sections.

4.3 Overview of the Gemini Descriptor Code

The Gemini descriptors provide access to keywords available in the headers of Gemini data.

The Gemini descriptor code is located in the `gemini_python` package in the `astrodata_Gemini/ADCONFIG_Gemini/descriptors` directory. When writing and adding new Gemini descriptors, a developer will require knowledge of the following files:

- `calculatorIndex.GEMINI.py`
- `CalculatorInterface_GEMINI.py`
- `GEMINI_Descriptors.py`
- `GEMINI_Keywords.py`
- `<INSTRUMENT>/<INSTRUMENT>_Descriptors.py`
- `<INSTRUMENT>/<INSTRUMENT>_Keywords.py`

The following files are required to create the `CalculatorInterface_GEMINI.py` file:

- `astrodata/scripts/mkCalculatorInterface`
- `DescriptorsList_GEMINI.py`
- `docstrings.py`

4.4 Introduction to the Gemini Descriptor Code

When a descriptor is called (as described in the *Basic Descriptor Usage* section and the *Advanced Descriptor Usage* section), the CalculatorInterface class (`CI`) is accessed, which is either contained in the module `CalculatorInterface_GEMINI.py` or, if this module doesn't exist, is automatically generated and stored in memory. The `CI` contains a function for each available descriptor (see *Appendix B* for an example function). These functions attempt to determine a value for the descriptor by performing the following steps:

- First, depending on the AstroData Type of the AstroData object, a single set of keywords and a single set of descriptor functions are determined via inheritance from the keyword files (`<INSTRUMENT>_Keywords.py`, `GEMINI_Keywords.py` and `FITS_Keywords.py`) and the descriptor files (`<INSTRUMENT>_Descriptor.py`, `GEMINI_Descriptor.py` and `FITS_Descriptor.py`), respectively, with the files at the start of each list taking precedence over files later in the list.
- If a descriptor function exists in the single set of descriptor functions determined in the first step for the descriptor being called, then that descriptor function is used to determine the value of the descriptor.
- If no descriptor function is found, the value of a keyword in the PHU of the AstroData object is returned as the value of the descriptor, where the keyword is the one directly associated to the descriptor and is contained in the single set of keywords determined in the first step.
- If a value for the descriptor is returned, either from a descriptor function or directly from the PHU of the AstroData object, a DescriptorValue (`DV`) object is instantiated (which contains the value of the descriptor) and returns this to the user.
- If a value for the descriptor is not found, an exception is raised (see the *Descriptor Exceptions* section).

A descriptor function is necessary in the cases where the descriptor does more than simply access a single keyword from the PHU of the AstroData object, e.g., if a descriptor requires access to multiple keywords, requires access to keywords in the pixel data extensions (a dictionary must be returned by the descriptor function, where the key is the ("EXTNAME", EXTVER) tuple) and / or requires some validation.

4.5 Description of the files used in the Gemini Descriptor Code

4.5.1 calculatorIndex.GEMINI.py

The calculatorIndex.GEMINI.py file contains a Python dictionary named calculatorIndex and is used to define which Python object (i.e., the descriptor class that defines the descriptor functions, in the form <module_name>.<calculator_class_name>) to use as the calculator for a given <INSTRUMENT>:

```
calculatorIndex = {
  "<INSTRUMENT>":"<INSTRUMENT>_Descriptors.<INSTRUMENT>_DescriptorCalc()",
  }
```

When adding descriptors for a new, undefined instrument, an appropriate entry must be added to the calculatorIndex dictionary in the calculatorIndex.GEMINI.py file as shown above.

4.5.2 CalculatorInterface_GEMINI.py

The CalculatorInterface_GEMINI.py module contains the CalculatorInterface class (CI), which contains a function for each available descriptor (see *Appendix B* for an example function). The CalculatorInterface_GEMINI.py module is automatically generated by the mkCalculatorInterface script, which is located in astrodata/scripts directory. Therefore, the CalculatorInterface_GEMINI.py module should never be edited directly. If the CalculatorInterface_GEMINI.py module does not exist, the CI is automatically generated and stored in memory.

4.5.3 GEMINI_Keywords.py

The Gemini specific keyword file GEMINI_Keywords.py contains a Python dictionary named GEMINI_KeyDict, where the key is a variable in the form key_<descriptor> and the value is the keyword directly associated to the descriptor <descriptor>:

```
GEMINI_KeyDict = {
  "key_airmass":"AIRMASS",
  ...
  "key_camera":"CAMERA",
  ...
  }
```

As shown above, the AIRMASS keyword is associated to the airmass descriptor via the variable key_airmass. When the airmass descriptor is called, the value of the single keyword AIRMASS in the PHU of the AstroData object is returned.

If a descriptor function in the Gemini specific descriptor file GEMINI_Descriptors.py requires access to additional keywords, appropriate variables must be defined in the GEMINI_KeyDict dictionary, so that keyword names are not hard-coded in the descriptor files, allowing a single variable to be used consistently by multiple descriptor functions:

```
GEMINI_KeyDict = {
  ...
  "key_pwfs1":"PWFS1_ST",
  ...
  }
```

4.5.4 GEMINI_Descriptors.py

The Gemini specific descriptor file `GEMINI_Descriptors.py` contains descriptor functions that apply to all Gemini data.

4.5.5 <INSTRUMENT>_Keywords.py

The instrument specific keyword files `<INSTRUMENT>_Keywords.py`, which are located in the corresponding `<INSTRUMENT>` directory in the `astrodata_Gemini/ADCONFIG_Gemini/descriptors` directory, contain a Python dictionary named `<INSTRUMENT>_KeyDict`, where the key is a variable in the form `key_<descriptor>` and the value is the keyword directly associated to the descriptor <descriptor>, and is used to overwrite (via inheritance) any of the Gemini specific keywords directly associated with the descriptor as defined in `GEMINI_Keywords.py`:

```
GMOS_KeyDict = {
  ...
  "key_camera":"INSTRUME",
  ...
  }
```

For GMOS data, the value of the single keyword `INSTRUME` in the PHU of the AstroData object is returned when the `camera` descriptor is called, instead of the value of the single keyword `CAMERA`, as defined in the Gemini specific keyword file `GEMINI_Keywords.py`.

If a descriptor function in the instrument specific descriptor files `<INSTRUMENT>_Descriptors.py` require access to additional keywords, appropriate variables must be defined in the `<INSTRUMENT>_KeyDict` dictionary, so that keyword names are not hard-coded in the descriptor files, allowing a single variable to be used consistently by multiple descriptor functions:

```
GMOS_KeyDict = {
  ...
  "key_ccdsum":"CCDSUM",
  ...
  }
```

4.5.6 <INSTRUMENT>_Descriptors.py

The instrument specific descriptor files `<INSTRUMENT>_Descriptors.py`, which are located in the corresponding `<INSTRUMENT>` directory in the `astrodata_Gemini/ADCONFIG_Gemini/descriptors` directory, contain descriptor functions that are specific to `<INSTRUMENT>` and are used to overwrite (via inheritance) any of the Gemini specific descriptor functions as defined in `GEMINI_Descriptors.py`. An example descriptor function (`detector_x_bin`) from `GMOS_Descriptors.py` can be found in *Appendix C*.

4.5.7 mkCalculatorInterface

The `mkCalculatorInterface` script is located in the `astrodata/scripts` directory and contains the code required to automatically generate the `CalculatorInterface_GEMINI.py` module. To create `CalculatorInterface_GEMINI.py`, run the following command in the `astrodata_Gemini/ADCONFIG_Gemini/descriptors` directory:

```
shell> mkCalculatorInterface > CalculatorInterface_GEMINI.py
```

The `mkCalculatorInterface` script uses the information in the `DescriptorsList_GEMINI.py` file and the `docstrings.py` file to create the `CalculatorInterface_GEMINI.py` module. This script should be run after making changes to the `DescriptorsList_GEMINI.py` file or the `docstrings.py` file.

4.5.8 DescriptorsList_GEMINI.py

The `DescriptorsList_GEMINI.py` file contains a list of Gemini descriptors and their default Python type:

```
[
  DD("airmass", pytype=float),
  ...
]
```

The `DescriptorsList_GEMINI.py` file is used by the `mkCalculatorInterface` script to generate a function for every descriptor in the `CalculatorInterface_Gemini.py` file.

4.5.9 docstrings.py

The `docstrings.py` file contains a function for every Gemini descriptor (as listed in the `DescriptorsList_GEMINI.py` file) where the docstring for each descriptor can be defined. The `docstrings.py` file is used by the `mkCalculatorInterface` script to include the docstrings for the descriptor functions in the `CalculatorInterface_Gemini.py` file.

4.6 How to add a new Gemini descriptor

The following instructions describe how to add a new descriptor to the system.

1. First, check to see whether the new descriptor has the same concept as a descriptor that already exists (*Appendix A*). If a new descriptor is required, edit the `DescriptorsList_GEMINI.py` file and add the new descriptor to the list in alphabetical order. Ensure that the default Python type for the descriptor is defined:

   ```
   [
     ...
     DD("<my_descriptor_name>", pytype=str),
     ...
   ]
   ```

2. Add a function with the same name as the new descriptor to the `docstrings.py` file and write a docstring for the new descriptor so that it can be included in the `CalculatorInterface_GEMINI.py` file.

3. Regenerate the `CalculatorInterface_GEMINI.py` file:

   ```
   shell> mkCalculatorInterface > CalculatorInterface_GEMINI.py
   ```

4. If the new descriptor is for a new, undefined <INSTRUMENT>, create an <INSTRUMENT> directory containing an <INSTRUMENT>_Descriptors.py file:

```
from GEMINI_Descriptors import GEMINI_DescriptorCalc
from <INSTRUMENT>_Keywords import <INSTRUMENT>_KeyDict

class <INSTRUMENT>_DescriptorCalc(GEMINI_DescriptorCalc):
    # Updating the global key dictionary with the local key dictionary
    # associated with this descriptor class
    _update_stdkey_dict = <INSTRUMENT>_KeyDict
```

   and an <INSTRUMENT>_Keywords.py file:

```
<INSTRUMENT>_KeyDict = {}
```

   In addition, add an appropriate entry to the calculatorIndex.Gemini.py file.

5. If the new descriptor simply returns the value of a single keyword in the PHU of the AstroData object, check whether the correct keyword is already defined in the GEMINI_KeyDict dictionary in the keyword file GEMINI_Keywords.py. If not, add an entry to the <INSTRUMENT>_KeyDict dictionary in the instrument specific keyword file <INSTRUMENT>_Keywords.py, specifying the new descriptor and the associated keyword:

```
<INSTRUMENT>_KeyDict = {
  "key_<my_descriptor_name>":"MYKEYWRD",
}
```

   The descriptor can now be tested; go to step 7.

6. If the new descriptor requires access to multiple keywords, requires access to keywords in the pixel data extensions and / or requires some validation, a descriptor function must be created. Depending on the type of information the new descriptor will provide, edit one of the following files to include the new descriptor function:

   - GEMINI_Descriptor.py
   - <INSTRUMENT>_Descriptor.py

   If access to a particular keyword is required, first check the keyword files (FITS_Keywords.py, GEMINI_Keywords.py and <INSTRUMENT>_Keywords.py) to see if it has already been defined. If required, the <INSTRUMENT>_Keywords.py file should be edited to contain any new keywords required for this new descriptor function.

7. Test the descriptor:

```
>>> from astrodata import AstroData
>>> ad = AstroData("N20091027S0137.fits")
>>> print ad.<my_descriptor_name>()
```

4.7 Descriptor Coding Guidelines

When creating descriptor functions, the guidelines below should be followed:

1. Return value
   - The descriptors will return the correct value, regardless of the data processing status of the AstroData object.
   - The descriptors will not write keywords to the headers of the AstroData object or cache any information, since it is no effort to use the descriptors to obtain the correct value as and when it is required.

- The value of a descriptor can be written to the history, for information only.

2. Return value Python type
   - The descriptors will always return a `DV` object to the user.
   - The `DV` object is instantiated by the `CI` for descriptors that obtain their values directly from the headers of the AstroData object. For descriptors that obtain their values from the descriptor functions, the descriptor functions should be coded to return a `DV` object. The `DV` object contains information related to the descriptor, including the value of the descriptor, the default Python type for that descriptor and the units of the descriptor.
3. Keyword access
   - The `phu_get_key_value` and `get_key_value` AstroData member functions should be used in the descriptor functions to access keywords in the PHU and the headers of the pixel data extensions, respectively, of an AstroData object.
4. Logging
   - Descriptors will not log any messages.
5. Raising exceptions
   - If the value of a descriptor can not be determined for whatever reason, the descriptor function should raise an exception.
   - The descriptor functions should never be coded to return None. Instead, a descriptor function should throw an exception with a message explaining why a value could not be returned (e.g., if the concept does not directly apply to the data). An exception thrown from a descriptor function will be caught by the `CI`.
6. Exception rule
   - Descriptors should throw exceptions on fatal errors.
   - Exceptions thrown on fatal errors (e.g., if a descriptor function is not found in a loaded calculator) should never be caught by the `CI`. The high level code, such as a script or a primitive, should catch any relevant exceptions.
7. Descriptor names
   - Descriptor names will be:
     - all lower case
     - terms separated with "_"
     - not instrument specific
     - not mode specific, mostly
   - A descriptor should describe a particular concept and apply for all instrument modes.
8. Standard parameters
   - Descriptors accept parameters, some with general purposes are standardized.
   - It is especially important for descriptor parameters to follow the Standard Parameter Names (http://gdpsg.wikis-internal.gemini.edu/index.php/GDPSG-NamingConventions#Standard_Parameter_Names) as they are front-facing to the user and should therefore be consistent.

4.8 Descriptor Exceptions

When writing descriptor functions, if a descriptor is unable to return a value, an exception should be raised in the code with an appropriate, explicit error message, so that it is clear to the user exactly why a value could not be returned. The exception information is stored in `exception_info` by the `CI` so that a user can access that information, if they wish to do so. A list of descriptor exceptions can be found in the `astrodata/Errors.py` module. These descriptor exceptions inherit from the DescriptorError base class and are caught by the `CI`. If the default setting of `throwExceptions = False` (line 62 in `astrodata/Calculator.py`), `None` is returned. During development, the developer should set `throwExceptions = True` so that exceptions are thrown.

APPENDIX

A

A COMPLETE LIST OF AVAILABLE DESCRIPTORS

Descriptors are designed such that essential keyword values that describe a particular concept can be accessed from the headers of any dataset in a consistent manner, regardless of which instrument was used to obtain the data. This is particularly useful for Gemini data, since the majority of keywords used to describe a particular concept at Gemini are not uniform between the instruments.

`airmass`

- the mean airmass of the observation

`amp_read_area`

- the composite string containing the name of the array amplifier and the readout area of the array used for the observation

`array_section`

- the unbinned section (in the form of a Python list of integers that uses 0-based indexing as default) of the array that was used to observe the data

`azimuth`

- the azimuth (in degrees between 0 and 360) of the observation

`camera`

- the camera used for the observation

`cass_rotator_pa`

- the cassegrain rotator position angle (in degrees between -360 and 360) of the observation

`central_wavelength`

- the central wavelength (in meters as default) of the observation

`coadds`

- the number of coadds used for the observation

`data_label`

- the unique identifying name (e.g., GN-2003A-C-2-52-003) of the observation

`data_section`

- the section (in the form of a Python list of integers that uses 0-based indexing as default) of the pixel data extensions that contains the data observed

`decker`

- the decker position used for the observation

`dec`

- the declination (in decimal degrees) of the observation

`detector_name`

- the name of each array used for the observation

`detector_roi_setting`

- the human-readable description of the detector Region Of Interest (ROI) setting (either 'Full Frame', 'CCD2', 'Central Spectrum', 'Central Stamp', 'Custom', 'Undefined' or 'Fixed'), which corresponds to the name of the ROI in the OT

`detector_rois_requested`

- the requested detector Region Of Interest (ROI)s of the observation

`detector_section`

- the unbinned section (in the form of a Python list of integers that uses 0-based indexing as default) of the detector that was used to observe the data

`detector_x_bin`

- the binning of the x-axis of the detector used for the observation

`detector_y_bin`

- the binning of the y-axis of the detector used for the observation

`disperser`

- the disperser used for the observation - difference between disperser/grating/prism? when should someone use disperser? **********

`dispersion_axis`

- the dispersion axis (along rows, x = 1; along columns, y = 2; along planes, z = 3) of the observation.

`dispersion`

- the dispersion (in meters per pixel as default) of the observation

`elevation`

- the elevation (in degrees) of the observation

`exposure_time`

- the total exposure time (in seconds) of the observation

`filter_name`

- the unique filter name identifier string used for the observation; when multiple filters are used, the filter names are concatenated with an ampersand

`focal_plane_mask`

- the focal plane mask used for the observation

`gain`

- the gain (in electrons per ADU) of the observation

`gain_setting`

- the gain setting (either 'high' or 'low') of the observation

`grating`

- the grating used for the observation

`group_id`

- the unique string that describes which stack a dataset belongs to; it is based on the observation_id

`instrument`

- the instrument used for the observation

`local_time`

- the local time (in HH:MM:SS.S) at the start of the observation

`mdf_row_id`

- the corresponding reference row in the Mask Definition File (MDF)

`nod_count`

- the number of nod and shuffle cycles in the nod and shuffle observation

`nod_pixels`

- the number of pixel rows the charge is shuffled by in the nod and shuffle observation

`nominal_atmospheric_extinction`

- the nominal atmospheric extinction (defined as coeff * (airmass - 1.0), where coeff is the site and filter specific nominal atmospheric extinction coefficient) of the observation

`nominal_photometric_zeropoint`

- the nominal photometric zeropoint of the observation

`non_linear_level`

- the non linear level in the raw images (in ADU) of the observation

`object`

- the name of the target object observed

`observation_class`

- the class (either 'science', 'progCal', 'partnerCal', 'acq', 'acqCal' or 'dayCal') of the observation

`observation_epoch`

- the epoch (in years) at the start of the observation

`observation_id`

- the ID (e.g., GN-2011A-Q-123-45) of the observation; it is used by group_id

`observation_type`

- the type (either 'OBJECT', 'DARK', 'FLAT', 'ARC', 'BIAS' or 'MASK') of the observation

`overscan_section`

- the section (in the form of a Python list of integers that uses 0-based indexing as default) of the pixel data extensions that contains the overscan data

`pixel_scale`

- the pixel scale (in arcsec per pixel) of the observation

`prism`

- the prism used for the observation

`program_id`

- the Gemini program ID (e.g., GN-2011A-Q-123) of the observation

`pupil_mask`

- the pupil mask used for the observation

`qa_state`

- the quality assessment state (either 'Undefined', 'Pass', 'Usable', 'Fail' or 'CHECK') of the observation

`ra`

- the Right Ascension (in decimal degrees) of the observation

`raw_bg`

- the raw background (as an integer percentile value) of the observation

`raw_cc`

- the raw cloud cover (as an integer percentile value) of the observation

`raw_iq`

- the raw image quality (as an integer percentile value) of the observation

`raw_wv`

- the raw water vapour (as an integer percentile value) of the observation

`read_mode`

- the read mode (either 'Very Faint Object(s)', 'Faint Object(s)', 'Medium Object', 'Bright Object(s)', 'Very Bright Object(s)', 'Low Background', 'Medium Background', 'High Background' or 'Invalid') of the observation

`read_noise`

- the estimated readout noise (in electrons) of the observation

`read_speed_setting`

- the read speed setting (either 'fast' or 'slow') of the observation

`requested_bg`

- the requested background (as an integer percentile value) of the observation

`requested_cc`

- the requested cloud cover (as an integer percentile value) of the observation

`requested_iq`

- the requested image quality (as an integer percentile value) of the observation

`requested_wv`

- the requested water vapour (as an integer percentile value) of the observation

`saturation_level`

- the saturation level (in ADU) of the observation

`slit`

- the name of the slit used for the observation

`telescope`

- the telescope used for the observation

`ut_date`

- the UT date (as a datetime object) at the start of the observation

`ut_datetime`

- the UT date and time (as a datetime object) at the start of the observation

`ut_time`

- the UT time (as a datetime object) at the start of the observation

`wavefront_sensor`

- the wavefront sensor (either 'AOWFS', 'OIWFS', 'PWFS1', 'PWFS2', some combination in alphabetic order separated with an ampersand or None) used for the observation

`wavelength_band`

- the wavelength band name (e.g., J, V, R, N) of the observation

`wavelength_reference_pixel`

- the 1-based reference pixel of the central wavelength of the observation

`well_depth_setting`

- the well depth setting (either 'Shallow', 'Deep' or 'Invalid') of the observation

`x_offset`

- the telescope offset in x (in arcsec) of the observation

`y_offset`

- the telescope offset in y (in arcsec) of the observation

APPENDIX

B

AN EXAMPLE FUNCTION FROM CALCULATORINTERFACE_GEMINI.PY

The example function below is auto-generated by the `mkCalculatorInterface` script. The `CalculatorInterface_GEMINI.py` file should never be edited directly.

```
def airmass(self, format=None, **args):
    """
    Return the airmass value

    :param dataset: the data set
    :type dataset: AstroData
    :param format: the return format
    :type format: string
    :rtype: float as default (i.e., format=None)
    :return: the mean airmass of the observation
    """
    try:
        self._lazyloadCalculator()
        keydict = self.descriptor_calculator._specifickey_dict
        key = "key_airmass"
        keyword = None
        if key in keydict.keys():
            keyword = keydict[key]

        if not hasattr(self.descriptor_calculator, "airmass"):
            if keyword is not None:
                retval = self.phu_get_key_value(keyword)
                if retval is None:
                    if hasattr(self, "exception_info"):
                        raise Errors.DescriptorError(self.exception_info)
            else:
                msg = ("Unable to find an appropriate descriptor "
                       "function or a default keyword for airmass")
                raise Errors.DescriptorError(msg)
        else:
            try:
                retval = self.descriptor_calculator.airmass(self, **args)
            except Exception as e:
                raise Errors.DescriptorError(e)


        ret = DescriptorValue( retval,
                               format = format,
                               name = "airmass",
                               keyword = keyword,
```

```
                            ad = self,
                            pytype = float )
        return ret

    except Errors.DescriptorError:
        if self.descriptor_calculator.throwExceptions == True:
            raise
        else:
            if not hasattr(self, \"exception_info\"):
                setattr(self, \"exception_info\", sys.exc_info()[1])
            return None
    except:
        raise
```

APPENDIX

C

AN EXAMPLE DESCRIPTOR FUNCTION FROM GMOS_DESCRIPTOR.PY

```
from astrodata import Errors
from GMOS_Keywords import GMOS_KeyDict
from GEMINI_Descriptors import GEMINI_DescriptorCalc

class GMOS_DescriptorCalc(GEMINI_DescriptorCalc):
    # Updating the global key dictionary with the local key dictionary
    # associated with this descriptor class
    _update_stdkey_dict = GMOS_KeyDict

    def __init__(self):
        GEMINI_DescriptorCalc.__init__(self)

    def detector_x_bin(self, dataset, **args):
        # Since this descriptor function accesses keywords in the headers of
        # the pixel data extensions, always return a dictionary where the key
        # of the dictionary is an (EXTNAME, EXTVER) tuple
        ret_detector_x_bin = {}

        # Determine the ccdsum keyword from the global keyword dictionary
        keyword = self.get_descriptor_key("key_ccdsum")

        # Get the value of the ccdsum keyword from the header of each pixel
        # data extension as a dictionary
        ccdsum_dict = gmu.get_key_value_dict(dataset, keyword)

        if ccdsum_dict is None:
            # The get_key_value_dict() function returns None if a value
            # cannot be found and stores the exception info. Re-raise the
            # exception. It will be dealt with by the CalculatorInterface.
            if hasattr(dataset, "exception_info"):
                raise dataset.exception_info

        for ext_name_ver, ccdsum in ccdsum_dict.iteritems():
            if ccdsum is None:
                detector_x_bin = None
            else:
                # Use the binning of the x-axis integer as the value
                detector_x_bin, detector_y_bin = ccdsum.split()

            # Update the dictionary with the binning of the x-axis value
            ret_detector_x_bin.update({ext_name_ver:detector_x_bin})

        return ret_detector_x_bin
```