
A User's Guide to Descriptors

Release 0.1

**Emma Hogan, Data Processing Software Group, Gemini
Observatory**

October 12, 2011

Contents

1	What are Descriptors?	1
2	Basic Descriptor Usage	3
3	Advanced Descriptor Usage	5
3.1	DescriptorValue	5
3.2	DescriptorUnits	7
4	Writing and Adding New Descriptors	9
4.1	Introduction to the Gemini Descriptor Code	9
4.2	Overview of the Gemini Descriptor Code	9
4.3	How to add a new descriptor	13
4.4	Descriptor Coding Guidelines	14
4.5	Descriptor Exceptions	15
4.6	Descriptor Docstrings	15
A	A Complete List of Available Descriptors	17
B	mkCalculatorInterface.py	19
C	An Example Function from CalculatorInterface.py	25
D	StandardDescriptorKeyDict.py	27
E	An Example Descriptor Function from GMOS_Descriptor.py	29

Chapter 1

What are Descriptors?

Descriptors are designed such that essential keyword values that describe a particular concept can be accessed from the headers of a given 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 2

Basic Descriptor Usage

The command `typewalk -l` lists all available descriptors. As of August 3, 2011, there are 61 descriptors available ([Appendix A](#)).

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

```
>>> from astrodatab import AstroData
# Load the fits file into AstroData
>>> ad = AstroData("N20091027S0137.fits")
# 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 science extension
>>> for ext in ad["SCI"]:
...     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 (the keywords themselves only exist in the PHU) and so only one value is returned. However, the gain applies to the pixel data extensions and so for this AstroData object, three values are returned, since there are three pixel data extensions. In this case, a Python dictionary is used to store the values, where the key of the dictionary is the (“EXTNAME“, EXTVER) tuple.

Chapter 3

Advanced Descriptor Usage

3.1 DescriptorValue

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

```
>>> print type(airmass)
<type 'instance'>
```

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'>
```

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 but 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.

The descriptor value 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 its default Python type. For example, when it is necessary to use an actual descriptor value (e.g., string, float, etc.) rather than a DV object as a key to a Python dictionary, the DV object can be cast to its 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'>
```

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}
```

A descriptor that is related to the pixel data extensions 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)
<type 'instance'>
>>> 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 value. Basic implementation of this feature has been completed and development is ongoing.

Chapter 4

Writing and Adding New Descriptors

4.1 Introduction to the Gemini Descriptor Code

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 user / developer will require knowledge of the following files:

- `CalculatorInterface.py`
- `mkCalculatorInterface.py`
- `StandardDescriptorKeyDict.py`
- `StandardGenericKeyDict.py`
- `StandardGEMINIKeyDict.py`
- `Standard<INSTRUMENT>KeyDict.py`
- `Generic_Descriptor.py`
- `GEMINI_Descriptor.py`
- `<INSTRUMENT>_Descriptor.py`
- `calculatorIndex.Gemini.py`

4.2 Overview of 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.py` file is accessed. This file contains a function for each available descriptor (see *Appendix C* for an example function), which first look directly in the PHU of the `AstroData` object for the associated descriptor keyword as defined in one of the `AstroData` standard key dictionary files (`Standard<INSTRUMENT>KeyDict.py`, `StandardGEMINIKeyDict.py`, `StandardGenericKeyDict.py` or `StandardDescriptorKeyDict.py`). The keywords in these dictionaries are combined via inheritance to create a final dictionary containing a single set of keywords associated with a particular `AstroData` object. The value of the keyword in the header of the data is returned as the value of the descriptor. If the associated descriptor keyword is not found in the header of the data, the descriptor files are then searched in the order below to attempt to find an appropriate descriptor function, which will return the value of the descriptor.

- `<INSTRUMENT>_Descriptor.py`

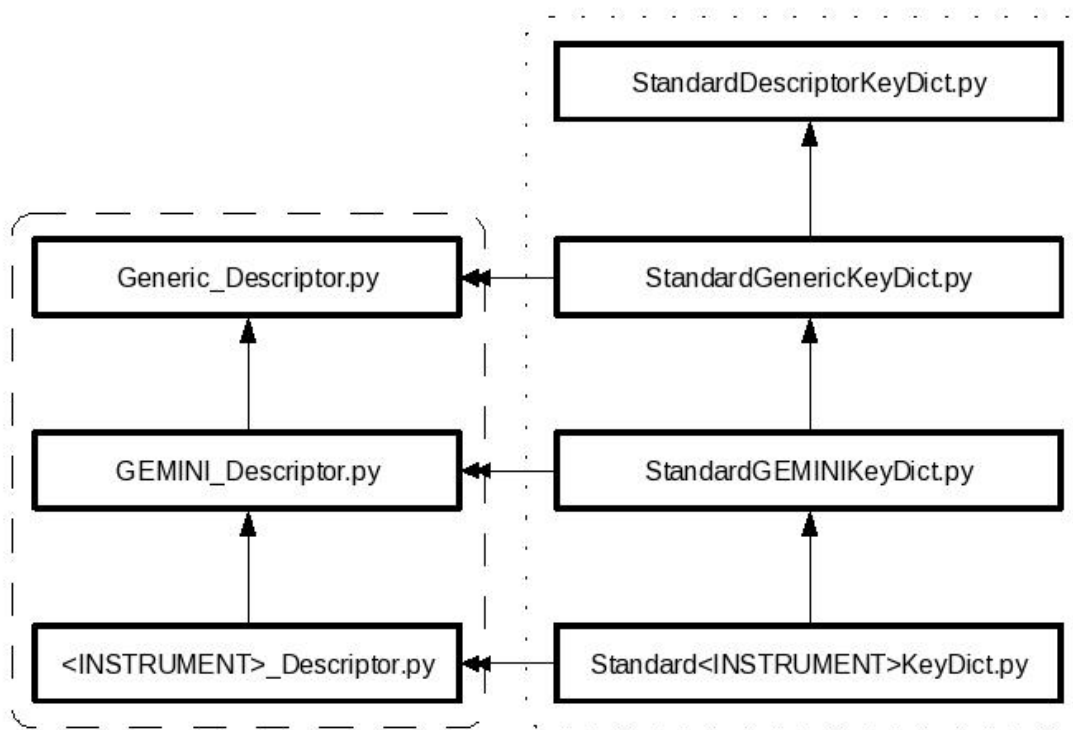


Figure 4.1: An overview showing the relationship between the descriptor files. The files located in the dashed box on the left are the descriptor files that contain descriptor functions. The files located in the dotted box on the right are the keyword files that contain dictionaries describing the one-to-one relationship between a descriptor / variable that is used in the associated descriptor files on the left (as shown by the double headed arrows) and the keyword associated with that descriptor / variable. In addition, the files within both the dashed box and the dotted box are subject to inheritance, i.e., any information contained within the files located lower in the chart will overwrite equivalent information that is contained within files located higher in the chart. The result of this inheritance is a single set of descriptor functions and keywords that will be used for a given AstroData object.

- `GEMINI_Descriptor.py`
- `Generic_Descriptor.py`

A descriptor function is used if a descriptor requires access to multiple keywords, requires access to keywords in the pixel data extensions (a dictionary must be created) and / or requires some validation. If no appropriate descriptor function is found, an exception is raised ([Section 4.5](#)). If a descriptor value is returned, either directly from the header of the data or from a descriptor function, the CI instantiates the DV object (which contains the descriptor value) and returns this to the user.

4.2.1 `CalculatorInterface.py`

The `CalculatorInterface.py` file contains the `CalculatorInterface` (CI) class, which contains a member function for each available descriptor (see [Appendix C](#) for an example function). This file is auto-generated by the `mkCalculatorInterface.py` file ([Appendix B](#)) and should never be edited directly.

4.2.2 `mkCalculatorInterface.py`

The `mkCalculatorInterface.py` file contains the code required to generate the file `CalculatorInterface.py`. To create `CalculatorInterface.py`, run the following command:

```
shell> python mkCalculatorInterface.py > CalculatorInterface.py
```

4.2.3 `StandardDescriptorKeyDict.py`

The `StandardDescriptorKeyDict.py` file contains a Python dictionary named `globalStdkeyDict`, which describes the one-to-one relationship between a descriptor and the AstroData standard keyword associated with that descriptor ([Appendix D](#)).

4.2.4 `StandardGenericKeyDict.py`

The `StandardGenericKeyDict.py` file contains a Python dictionary named `stdkeyDictGeneric`, which describes the one-to-one relationship between a descriptor / variable that is used in the generic descriptor file `Generic_Descriptor.py` and the keyword associated with that descriptor / variable. The values in this dictionary overwrite (via inheritance) the AstroData standard values defined in `StandardDescriptorKeyDict.py`.

4.2.5 `StandardGEMINIKeyDict.py`

The `StandardGEMINIKeyDict.py` file contains a Python dictionary named `stdkeyDictGEMINI` and is used to overwrite (via inheritance) any of the AstroData standard keywords defined in `StandardDescriptorKeyDict.py` and any of the generic keywords defined in `StandardGenericKeyDict.py`. It is also used to define variables for any keywords that need to be accessed in the Gemini specific descriptor file `GEMINI_Descriptor.py`.

4.2.6 `Standard<INSTRUMENT>KeyDict.py`

The `Standard<INSTRUMENT>KeyDict.py` files contain a Python dictionary named `stdkeyDict<INSTRUMENT>` and is used to overwrite (via inheritance) any of the AstroData

standard keywords defined in `StandardDescriptorKeyDict.py`, any of the generic keywords defined in `StandardGenericKeyDict.py` and any of the Gemini specific keyword defined in `StandardGEMINIKeyDict.py`. It is also used to define variables for any keywords that need to be accessed in the instrument specific descriptor file `<INSTRUMENT>_Descriptor.py`. These instrument specific files are located in the corresponding `<INSTRUMENT>` directory in the `astrodatab_Gemini/ADCONFIG_Gemini/descriptors` directory.

4.2.7 `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>`. An example of this dictionary is shown below.

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

4.2.8 `Generic_Descriptor.py`

The generic descriptor file `Generic_Descriptor.py` contains descriptor functions describing 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 4 descriptors currently available that access these FITS standard keywords:

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

4.2.9 `GEMINI_Descriptor.py`

The Gemini specific descriptor file `GEMINI_Descriptor.py` contains descriptor functions describing those keywords that are standard within Gemini. There are currently 142 Gemini standard keywords, which are relevant to the data file as a whole and exist in the PHU of the data. There are 21 descriptors currently available that access these Gemini standard keywords:

- `airmass` [`AIRMASS`]
- `azimuth` [`AZIMUTH`]
- `cass_rotator_pa` [`CRPA`]
- `data_label` [`DATALAB`]
- `dec` [`DEC`]
- `elevation` [`ELEVATIO`]
- `local_time` [`LT`]
- `observation_class` [`OBSCLASS`]
- `observation_id` [`OBSID`]

- `observation_type` [OBSTYPE]
- `observation_epoch` [OBSEPOCH]
- `program_id` [GEMPRGID]
- `qa_state` [RAWPIREQ, RAWGEMQA]
- `ra` [RA]
- `raw_bg` [RAWBG]
- `raw_cc` [RAWCC]
- `raw_iq` [RAWIQ]
- `raw_wv` [RAWWV]
- `ut_time` [UT]
- `wavefront_sensor` [OIWFS_ST, PWFS2_ST, AOWFS_ST]
- `x_offset` [XOFFSET]
- `y_offset` [YOFFSET]

4.2.10 <INSTRUMENT>_Descriptor.py

The instrument specific descriptor files `<INSTRUMENT>_Descriptor.py` contain descriptor functions specific to that `<INSTRUMENT>`. These instrument specific files are located in the corresponding `<INSTRUMENT>` directory in the `astrodatab_Gemini/ADCONFIG_Gemini/descriptors` directory.

4.3 How to add a new descriptor

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

1. First, check to see whether a descriptor already exists that has the same concept as the new descriptor to be added ([Appendix A](#)). If a new descriptor is required, edit the `mkCalculatorInterface.py` file and add the new descriptor to the DD constructor in the descriptors list in alphabetical order. Ensure that the default Python type for the descriptor is defined:

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

2. Regenerate the `CalculatorInterface.py` file:

```
shell> python mkCalculatorInterface.py > CalculatorInterface.py
```

3. Edit the `globalStdkeyDict` dictionary in the `StandardDescriptorKeyDict.py` file to include the AstroData standard keyword associated with the new descriptor ([Appendix D](#)).
4. If the new descriptor simply requires access to the AstroData standard keyword in the header of the data and returns the value, the descriptor can now be tested; go to step 7.

However, if the new descriptor requires access to a keyword that is different from the AstroData standard keyword (perhaps specific to a particular <INSTRUMENT>), go to step 5. If the new descriptor requires access to multiple keywords and / or requires some validation, a descriptor function must be created; go to step 6.

5. If the new descriptor requires access to a keyword that is different from the AstroData standard keyword, edit either the `StandardGenericKeyDict.py` file, the `StandardGEMINIKeyDict.py` file or the `Standard<INSTRUMENT>KeyDict.py` file (as appropriate) to include the keyword associated with the new descriptor. The descriptor can now be tested; go to step 7.
6. If the new descriptor requires access to multiple keywords and / or requires some validation, a descriptor function should 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:

- `Generic_Descriptor.py`
- `GEMINI_Descriptor.py`
- `<INSTRUMENT>_Descriptor.py`

The `<INSTRUMENT>_Descriptor.py` descriptor file is located in the `<INSTRUMENT>` directory. If the new descriptor is for a new `<INSTRUMENT>`, create an `<INSTRUMENT>` directory and edit the `calculatorIndex.Gemini.py` file appropriately ([Section 4.2.7](#)). An example descriptor function (`detector_x_bin`) from `GMOS_Descriptor.py` can be found in [Appendix E](#). If the descriptor should return more than one value, i.e., one value for each pixel data extension, a dictionary should be returned by the descriptor function, where the key is the ("EXTNAME", EXTVER) tuple. If access to a particular keyword is required, first check the appropriate keyword files (`StandardDescriptorKeyDict.py`, `StandardGenericKeyDict.py`, `StandardGEMINIKeyDict.py` and `Standard<INSTRUMENT>KeyDict.py`) to see if it has already been defined. If required, the `Standard<INSTRUMENT>KeyDict.py` file should be edited to contain any new keywords required for this new descriptor function.

7. Test the descriptor:

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

4.4 Descriptor Coding Guidelines

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

- The descriptors determine the correct keyword value for data at any point during processing
- The descriptors will not write keywords to the headers of the data; the descriptors only obtain information
- The descriptor functions themselves do not log any messages
- The descriptor values are not cached, since it is no effort to use the descriptors to obtain the correct value as and when it is required
- The descriptor values can be written to the history, for information only
- The `phu_get_key_value` and `get_key_value` AstroData member functions are used in the descriptor functions to access keywords in the headers of the AstroData objects

For example, for raw GMOS data, the `gain` descriptor uses the raw keywords in the header and a look up table to determine the gain value. During the first data processing step for Gemini data (which includes standardizing the headers of the data), the value of the raw `GAIN` keyword is overwritten (since it was incorrect in the raw data) and the value of the raw `GAIN` keyword is written to the `HISTORY` keyword (for information only). If a descriptor is then

called after the first processing step, the `gain` descriptor reads the value directly from the `GAIN` keyword. This way, keyword values are always correct, regardless of the processing state of the data (and any external system that wishes to work on that data will also access the correct values).

4.5 Descriptor Exceptions

Normally, if a descriptor is unable to return a value, `None` is returned instead. However, exceptions that describe exactly why a value could not be returned (where applicable) are stored so that a user can access that information, if they wish to do so.

When writing descriptor functions, exceptions should be raised in the code with an appropriate, explicit error message, so that it is clear to the user exactly what went wrong. The exception is caught by the `CI` and if `throwExceptions = False` (line 62 in `astrodata/Calculator.py`), the exception information is stored in `exception_info` and `None` is returned. Otherwise the exception is thrown. During development, `throwExceptions = True` so that exceptions are thrown. When the code is released, `throwExceptions = False` and the exception information will be available in `exception_info`. Available `astrodata` exceptions can be found in `astrodata/Errors.py`. Additional required exceptions can be added to this file, if necessary.

4.6 Descriptor Docstrings

The current descriptor docstrings can be found in the member functions in the `CI`. These docstrings are generated by the `mkCalculatorInterface.py` file using the information located in the `descriptorDescriptionDict.py` file. It is likely that the information in the `descriptorDescriptionDict.py` file will be stored in the `DD` constructor in the future.

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 a given 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.

```
shell> typewalk -l
Available Descriptors
    airmass
    amp_read_area
    azimuth
    camera
    cass_rotator_pa
    central_wavelength
    coadds
    data_label
    data_section
    dec
    decker
    detector_section
    detector_x_bin
    detector_y_bin
    disperser
    dispersion
    dispersion_axis
    elevation
    exposure_time
    filter_name
    focal_plane_mask
    gain
    gain_setting
    grating
    group_id
    instrument
    local_time
    mdf_row_id
    nod_count
    nod_pixels
    non_linear_level
    object
    observation_class
```

observation_epoch
observation_id
observation_type
overscan_section
pixel_scale
prism
program_id
pupil_mask
qa_state
ra
raw_bg
raw_cc
raw_iq
raw_wv
read_mode
read_noise
read_speed_setting
saturation_level
slit
telescope
ut_date
ut_datetime
ut_time
wavefront_sensor
wavelength_reference_pixel
well_depth_setting
x_offset
y_offset

Appendix B

mkCalculatorInterface.py

```
from datetime import datetime
from descriptorDescriptionDict import asDictArgDict
from descriptorDescriptionDict import descriptorDescDict
from descriptorDescriptionDict import detailedNameDict
from descriptorDescriptionDict import stripIDArgDict

class DescriptorDescriptor:
    name = None
    description = None
    pytype = None
    unit = None

    thunkfuncbuff = """
def %(name)s(self, format=None, **args):
    \"\"\"
    %(description)s
    \"\"\"
    try:
        self._lazyloadCalculator()
        keydict = self.descriptor_calculator._specifickey_dict
        #print hasattr(self.descriptor_calculator, "%(name)s")
        if not hasattr(self.descriptor_calculator, "%(name)s"):
            key = "key_"+"%(name)s"
            #print "mkCI10:",key, repr(keydict)
            #print "mkCI12:", key in keydict
            if key in keydict.keys():
                retval = self.phu_get_key_value(keydict[key])
                if retval is None:
                    if hasattr(self, "exception_info"):
                        raise self.exception_info
            else:
                msg = "Unable to find an appropriate descriptor function "
                msg += "or a default keyword for %(name)s"
                raise KeyError(msg)
        else:
            retval = self.descriptor_calculator.%(name)s(self, **args)

    %(pytypeimport)s
    ret = DescriptorValue( retval,
                           format = format,
                           name = "%(name)s",
```

```
        ad = self,
        pytype = %(pytype)s )

    return ret
except:
    if not hasattr(self, "exception_info"):
        setattr(self, "exception_info", sys.exc_info()[1])
    if (self.descriptor_calculator is None
        or self.descriptor_calculator.throwExceptions == True):
        raise
    else:
        #print "NONE BY EXCEPTION"
        self.exception_info = sys.exc_info()[1]
        return None

"""
def __init__(self, name=None, pytype=None):
    self.name = name
    if pytype:
        self.pytype = pytype
        rtype = pytype.__name__
    try:
        desc = descriptorDescDict[name]
    except:
        if rtype == 'str':
            rtype = 'string'
        if rtype == 'int':
            rtype = 'integer'
        try:
            dname = detailedNameDict[name]
        except:
            dname = name
        try:
            asDictArg = asDictArgDict[name]
        except:
            asDictArg = 'no'
        try:
            stripIDArg = stripIDArgDict[name]
        except:
            stripIDArg = 'no'

        if stripIDArg == 'yes':
            desc = 'Return the %(name)s value\n' % {'name':name} + \
                '      :param dataset: the data set\n' + \
                '      :type dataset: AstroData\n' + \
                '      :param stripID: set to True to remove the ' + \
                'component ID from the \n' + \
                'returned %(name)s value\n' % {'name':name} + \
                '      :type stripID: Python boolean\n' + \
                '      :param pretty: set to True to return a ' + \
                'human meaningful \n' + \
                '                        %(name)s ' % {'name':name} + \
                'value\n' + \
                '      :type pretty: Python boolean\n' + \
                '      :rtype: %(rtype)s ' % {'rtype':rtype} + \
                'as default (i.e., format=None)\n' + \
                '      :return: the %(dname)s' \
                % {'dname':dname}
        elif asDictArg == 'yes':
            desc = 'Return the %(name)s value\n' % {'name':name} + \
```

```

        :param dataset: the data set\n' + \
        :type dataset: AstroData\n' + \
        :param format: the return format\n' + \
        set to as_dict to return a ' + \
'dictionary, where the number ' + \
'\n          of dictionary elements ' + \
'equals the number of pixel data ' + \
'\n          extensions in the image. ' + \
'The key of the dictionary is ' + \
'\n          an (EXTNAME, EXTVER) ' + \
'tuple, if available. Otherwise, ' + \
'\n          the key is the integer ' + \
'index of the extension.\n' + \
        :type format: string\n' + \
        :rtype: %(rtype)s ' % {'rtype':rtype} + \
'as default (i.e., format=None)\n' + \
        :rtype: dictionary containing one or more ' + \
'%(rtype)s(s) ' % {'rtype':rtype} + \
'(format=as_dict)\n' + \
        :return: the %(dname)s' \
% {'dname':dname}

    else:
        desc = 'Return the %(name)s value\n' % {'name':name} + \
        :param dataset: the data set\n' + \
        :type dataset: AstroData\n' + \
        :param format: the return format\n' + \
        :type format: string\n' + \
        :rtype: %(rtype)s ' % {'rtype':rtype} + \
'as default (i.e., format=None)\n' + \
        :return: the %(dname)s' \
        % {'dname':dname}

    self.description = desc

def funcbody(self):
    if self.pytype:
        pytypestr = self.pytype.__name__
    else:
        pytypestr = "None"
    if pytypestr == "datetime":
        pti = "from datetime import datetime"
    else:
        pti = ""
    #print "mkC150:", pti
    ret = self.thunkfuncbuff % {'name':self.name,
                               'pytypeimport': pti,
                               'pytype': pytypestr,
                               'description':self.description}

    return ret

DD = DescriptorDescriptor

descriptors = [ DD("airmass", pytype=float),
                DD("amp_read_area", pytype=str),
                DD("azimuth", pytype=float),
                DD("camera", pytype=str),
                DD("cass_rotator_pa", pytype=float),

```

```
DD("central_wavelength", pytype=float),
DD("coadds", pytype=int),
DD("data_label", pytype=str),
DD("data_section", pytype=list),
DD("dec", pytype=float),
DD("decker", pytype=str),
DD("detector_section", pytype=list),
DD("detector_x_bin", pytype=int),
DD("detector_y_bin", pytype=int),
DD("disperser", pytype=str),
DD("dispersion", pytype=float),
DD("dispersion_axis", pytype=int),
DD("elevation", pytype=float),
DD("exposure_time", pytype=float),
DD("filter_name", pytype=str),
DD("focal_plane_mask", pytype=str),
DD("gain", pytype=float),
DD("grating", pytype=str),
DD("group_id", pytype=str),
DD("gain_setting", pytype=str),
DD("instrument", pytype=str),
DD("local_time", pytype=str),
DD("mdf_row_id", pytype=int),
DD("nod_count", pytype=int),
DD("nod_pixels", pytype=int),
DD("non_linear_level", pytype=int),
DD("object", pytype=str),
DD("observation_class", pytype=str),
DD("observation_epoch", pytype=str),
DD("observation_id", pytype=str),
DD("observation_type", pytype=str),
DD("overscan_section", pytype=list),
DD("pixel_scale", pytype=float),
DD("prism", pytype=str),
DD("program_id", pytype=str),
DD("pupil_mask", pytype=str),
DD("qa_state", pytype=str),
DD("ra", pytype=float),
DD("raw_bg", pytype=str),
DD("raw_cc", pytype=str),
DD("raw_iq", pytype=str),
DD("raw_wv", pytype=str),
DD("read_mode", pytype=str),
DD("read_noise", pytype=float),
DD("read_speed_setting", pytype=str),
DD("saturation_level", pytype=int),
DD("slit", pytype=str),
DD("telescope", pytype=str),
DD("ut_date", pytype=datetime),
DD("ut_datetime", pytype=datetime),
DD("ut_time", pytype=datetime),
DD("wavefront_sensor", pytype=str),
DD("wavelength_reference_pixel", pytype=float),
DD("well_depth_setting", pytype=str),
DD("x_offset", pytype=float),
DD("y_offset", pytype=float),
```

```
]
```

```
wholeout = """import sys
from astrodatab import Descriptors
from astrodatab.Descriptors import DescriptorValue
from astrodatab import Errors

class CalculatorInterface:

    descriptor_calculator = None
%(descriptors)s
# UTILITY FUNCTIONS, above are descriptor thunks
    def _lazyloadCalculator(self, **args):
        '''Function to put at top of all descriptor members
        to ensure the descriptor is loaded. This way we avoid
        loading it if it is not needed.'''
        if self.descriptor_calculator is None:
            self.descriptor_calculator = Descriptors.get_calculator(self, **args)
"""
out = ""

for dd in descriptors:
    out += dd.funcbody()

finalout = wholeout % {"descriptors": out}

print finalout
```


Appendix C

An Example Function from CalculatorInterface.py

The example function below is auto-generated by the mkCalculatorInterface.py file. The CalculatorInterface.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
        if not hasattr(self.descriptor_calculator, "airmass"):
            key = "key_"+"airmass"
            if key in keydict.keys():
                retval = self.phu_get_key_value(keydict[key])
                if retval is None:
                    if hasattr(self, "exception_info"):
                        raise self.exception_info
            else:
                msg = "Unable to find an appropriate descriptor function "
                msg += "or a default keyword for airmass"
                raise KeyError(msg)
        else:
            retval = self.descriptor_calculator.airmass(self, **args)

        ret = DescriptorValue( retval,
                               format = format,
                               name = "airmass",
                               ad = self,
                               pytype = float )

    return ret
except:
```

```
if (self.descriptor_calculator is None
    or self.descriptor_calculator.throwExceptions == True):
    raise
else:
    #print "NONE BY EXCEPTION"
    self.exception_info = sys.exc_info()[1]
    return None
```

Appendix D

StandardDescriptorKeyDict.py

```
globalStdkeyDict = {
    "key_airmass": "AIRMASS",
    "key_amp_read_area": "AMPROA",
    "key_azimuth": "AZIMUTH",
    "key_camera": "CAMERA",
    "key_cass_rotator_pa": "CRPA",
    "key_central_wavelength": "CWAVE",
    "key_coadds": "COADDS",
    "key_data_label": "DATA LAB",
    "key_data_section": "DATA SEC",
    "key_dec": "DEC",
    "key_decker": "DECKER",
    "key_detector_section": "DET SEC",
    "key_detector_x_bin": "XCDBIN",
    "key_detector_y_bin": "YCDBIN",
    "key_disperser": "DISPERSR",
    "key_dispersion": "WDELTA",
    "key_dispersion_axis": "DISP AXIS",
    "key_elevation": "ELEVATIO",
    "key_exposure_time": "EXPTIME",
    "key_filter_name": "FILTNAME",
    "key_focal_plane_mask": "FPMASK",
    "key_gain": "GAIN",
    "key_gain_setting": "GAINSET",
    "key_grating": "GRATING",
    "key_group_id": "GROUPID",
    "key_instrument": "INSTRUME",
    "key_local_time": "LT",
    "key_mdf_row_id": "MDFROW",
    "key_nod_count": "NODCOUNT",
    "key_nod_pixels": "NODPIX",
    "key_non_linear_level": "NONLINEA",
    "key_object": "OBJECT",
    "key_observation_class": "OBSCCLASS",
    "key_observation_epoch": "OBSEPOCH",
    "key_observation_id": "OBSID",
    "key_observation_type": "OBSTYPE",
    "key_overscan_section": "OVERSSEC",
    "key_pixel_scale": "PIXSCALE",
    "key_prism": "PRISM",
    "key_program_id": "GEMPRGID",
```

```
"key_pupil_mask": "PUPILMSK",
"key_qa_state": "QASTATE",
"key_ra": "RA",
"key_raw_bg": "RAWBG",
"key_raw_cc": "RAWCC",
"key_raw_iq": "RAWIQ",
"key_raw_wv": "RAWWV",
"key_read_mode": "READMODE",
"key_read_noise": "RDNOISE",
"key_read_speed_setting": "RDSPDSET",
"key_saturation_level": "SATLEVEL",
"key_slit": "SLIT",
"key_telescope": "TELESCOP",
"key_ut_date": "DATE-OBS",
"key_ut_datetime": "DATETIME",
"key_ut_time": "UT",
"key_wavefront_sensor": "WFS",
"key_wavelength_reference_pixel": "WREFPIX",
"key_well_depth_setting": "WELDEPTH",
"key_x_offset": "XOFFSET",
"key_y_offset": "YOFFSET",
}
```


Appendix E

An Example Descriptor Function from GMOS_Descriptor.py

```
from astrodatab import Errors
from StandardGMOSKeyDict import stdkeyDictGMOS
from GEMINI_Descriptor 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 = stdkeyDictGMOS

    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 = {}

        # Loop over the science extensions in the dataset
        for ext in dataset["SCI"]:

            # Get the ccdsum value from the header of each pixel data
            # extension. The ccdsum keyword is defined in the local key
            # dictionary (stdkeyDictGMOS) but is read from the updated global
            # key dictionary (self.get_descriptor_key())
            ccdsum = ext.get_key_value(self.get_descriptor_key("key_ccdsum"))

            if ccdsum is None:
                # The get_key_value() 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(ext, "exception_info"):
                    raise ext.exception_info

        detector_x_bin, detector_y_bin = ccdsum.split()

        # Return a dictionary with the binning of the x-axis integer as
```

```
# the value
ret_detector_x_bin.update({
    (ext.extname(), ext.extver()):int(detector_x_bin)})

if ret_detector_x_bin == {}:
    # If the dictionary is still empty, the AstroData object was not
    # automatically assigned a "SCI" extension and so the above for
    # loop was not entered
    raise Errors.CorruptDataError()

return ret_detector_x_bin
```