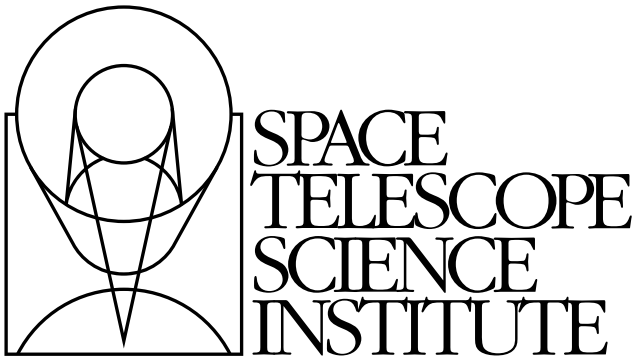

stsci.tools Documentation

Release 2.9

SSB

June 23, 2016

The STSCI.TOOLS package in STScI_Python provides many functions for use by multiple software packages.

Contents:

CHAPTER

ONE

GENERAL PYTHON UTILITIES

The modules and functions described here provide support for many of the most general operations used through out STScI_Python. fileutil.py – General file functions

These were initially designed for use with PyDrizzle. These functions only rely on booleans 'yes' and 'no', PyFITS and readgeis.

This file contains both IRAF-compatibility and general file access functions. General functions included are:

```
DEGTORAD(deg), RADTODEG(rad)

DIVMOD(num,val)

convertDate(date)
    Converts the DATE date string into a decimal year.

decimal_date(date-obs,time-obs=None)
    Converts the DATE-OBS (with optional TIME-OBS) string into a decimal year

buildRootname(filename, extn=None, extlist=None)

buildNewRootname(filename, ext=None)

parseFilename(filename)
    Splits a input name into a tuple containing (filename, group/extension)

getKeyword(filename, keyword, default=None, handle=None)

getHeader(filename,handle=None)
     Return a copy of the PRIMARY header, along with any group/extension
     header, for this filename specification.

getExtn(fimg,extn=None)
    Returns a copy of the specified extension with data from PyFITS object
    'fimg' for desired file.

updateKeyword(filename, key, value)

openImage(filename,mode='readonly',memmap=0,fitsname=None)
     Opens file and returns PyFITS object.
     It will work on both FITS and GEIS formatted images.

findFile(input)

checkFileExists(filename,directory=None)
```

```
removeFile(inlist):
    Utility function for deleting a list of files or a single file.

rAsciiLine(ifile)
    Returns the next non-blank line in an ASCII file.

readAsnTable(input,output=None,prodonly=yes)
    Reads an association (ASN) table and interprets inputs and output.
    The 'prodonly' parameter specifies whether to use products as inputs
        or not; where 'prodonly=no' specifies to only use EXP as inputs.

isFits(input) - returns (True|False, fitstype), fitstype is one of
                ('simple', 'mef', 'waiver')
```

IRAF compatibility functions (abbreviated list):

```
osfn(filename)
    Convert IRAF virtual path name to OS pathname

show(*args, **kw)
    Print value of IRAF or OS environment variables

time()
    Print current time and date

access(filename)
    Returns true if file exists, where filename can include IRAF variables
```

stsci.tools.fileutil.**DEGTORAD**(*deg*)

stsci.tools.fileutil.**DIVMOD**(*num*, *val*)

stsci.tools.fileutil.**Expand**(*instring*, *noerror=0*)
Expand a string with embedded IRAF variables (IRAF virtual filename).

Allows comma-separated lists. Also uses os.path.expanduser to replace '~' symbols.

Set the noerror flag to silently replace undefined variables with just the variable name or null (so Expand('abc$def') = 'abcdef' and Expand('(abc)def') = 'def'). This is the IRAF behavior, though it is confusing and hides errors.

stsci.tools.fileutil.**RADTODEG**(*rad*)

stsci.tools.fileutil.**access**(*filename*)
Returns true if file exists.

stsci.tools.fileutil.**buildFITSName**(*geisname*)
Build a new FITS filename for a GEIS input image.

stsci.tools.fileutil.**buildNewRootname**(*filename*, *extn=None*, *extlist=None*)
Build rootname for a new file.

Use 'extn' for new filename if given, does NOT append a suffix/extension at all.

Does NOT check to see if it exists already. Will ALWAYS return a new filename.

stsci.tools.fileutil.**buildRootname**(*filename*, *ext=None*)
Build a new rootname for an existing file and given extension.

Any user supplied extensions to use for searching for file need to be provided as a list of extensions.

Examples

```
>>> rootname = buildRootname(filename, ext=['_dth.fits'])
```

stsci.tools.fileutil.**buildRotMatrix**(*theta*)

stsci.tools.fileutil.**checkFileExists**(*filename*, *directory=None*)
Checks to see if file specified exists in current or specified directory.

Default is current directory. Returns 1 if it exists, 0 if not found.

stsci.tools.fileutil.**convertDate**(*date*)
Convert DATE string into a decimal year.

stsci.tools.fileutil.**copyFile**(*input*, *output*, *replace=None*)
Copy a file whole from input to output.

stsci.tools.fileutil.**countExtn**(*fimg*, *extname='SCI'*)
Return the number of 'extname' extensions, defaulting to counting the number of SCI extensions.

stsci.tools.fileutil.**decimal_date**(*dateobs*, *timeobs=None*)
Convert DATE-OBS (and optional TIME-OBS) into a decimal year.

stsci.tools.fileutil.**defvar**(*varname*)
Returns true if CL variable is defined.

stsci.tools.fileutil.**envget**(*var*, *default=None*)
Get value of IRAF or OS environment variable.

stsci.tools.fileutil.**findExtname**(*fimg*, *extname*, *extver=None*)
Returns the list number of the extension corresponding to EXTNAME given.

stsci.tools.fileutil.**findFile**(*input*)
Search a directory for full filename with optional path.

stsci.tools.fileutil.**findKeywordExtn**(*ft*, *keyword*, *value=None*)
This function will return the index of the extension in a multi-extension FITS file which contains the desired keyword with the given value.

stsci.tools.fileutil.**getDate**()
Returns a formatted string with the current date.

stsci.tools.fileutil.**getExtn**(*fimg*, *extn=None*)
Returns the PyFITS extension corresponding to extension specified in filename.

Defaults to returning the first extension with data or the primary extension, if none have data. If a non-existent extension has been specified, it raises a *KeyError* exception.

stsci.tools.fileutil.**getFilterNames**(*header*, *filternames=None*)
Returns a comma-separated string of filter names extracted from the input header (PyFITS header object). This function has been hard-coded to support the following instruments:

ACS, WFPC2, STIS

This function relies on the 'INSTRUME' keyword to define what instrument has been used to generate the observation/header.

The 'filternames' parameter allows the user to provide a list of keyword names for their instrument, in the case their instrument is not supported.

stsci.tools.fileutil.**getHeader**(*filename*, *handle=None*)
Return a copy of the PRIMARY header, along with any group/extension header for this filename specification.

stsci.tools.fileutil.**getKeyword**(*filename*, *keyword*, *default=None*, *handle=None*)
General, write-safe method for returning a keyword value from the header of a IRAF recognized image.

Returns the value as a string.

stsci.tools.fileutil.**getLTime**()
Returns a formatted string with the current local time.

stsci.tools.fileutil.**getVarDict**()
Returns dictionary all IRAF variables.

stsci.tools.fileutil.**getVarList**()
Returns list of names of all IRAF variables.

stsci.tools.fileutil.**help**()

stsci.tools.fileutil.**interpretDQvalue**(*input*)
Converts an integer 'input' into its component bit values as a list of power of 2 integers.

For example, the bit value 1027 would return [1, 2, 1024]

stsci.tools.fileutil.**isFits**(*input*)

Returns
isFits: tuple

An `(isfits, fitstype)` tuple. The values of `isfits` and `fitstype` are specified as:

- `isfits`: TruelFalse
- `fitstype`: if True, one of 'waiver', 'mef', 'simple'; if False, None

Notes

Input images which do not have a valid FITS filename will automatically result in a return of (False, None).

In the case that the input has a valid FITS filename but runs into some error upon opening, this routine will raise that exception for the calling routine/user to handle.

stsci.tools.fileutil.**listVars**(*prefix=''*, *equals='\t= '*, ***kw*)
List IRAF variables.

stsci.tools.fileutil.**openImage**(*filename*, *mode='readonly'*, *memmap=0*, *writefits=True*, *clobber=True*, *fitsname=None*)
Opens file and returns PyFITS object. Works on both FITS and GEIS formatted images.

Parameters
filename: str

name of input file

mode: str

mode for opening file based on PyFITS *mode* parameter values

memmap: int

switch for using memory mapping, 0 for no, 1 for yes

writefits: bool

if True, will write out GEIS as multi-extension FITS and return handle to that opened GEIS-derived MEF file

clobber: bool

overwrite previously written out GEIS-derived MEF file

fitsname: str

name to use for GEIS-derived MEF file, if None and writefits==True, will use 'buildFITSName()' to generate one

Notes

If a GEIS or waivered FITS image is used as input, it will convert it to a MEF object and only if `writefits = True` will write it out to a file. If `fitsname = None`, the name used to write out the new MEF file will be created using *buildFITSName*.

stsci.tools.fileutil.**osfn**(*filename*)
Convert IRAF virtual path name to OS pathname.

stsci.tools.fileutil.**parseExtn**(*extn=None*)
Parse a string representing a qualified fits extension name as in the output of *parseFilename* and return a tuple `(str(extname), int(extver))`, which can be passed to `astropy.io.fits` functions using the 'ext' kw.

Default return is the first extension in a fits file.

Examples

```
>>> parseExtn('sci, 2')
('sci', 2)
>>> parseExtn('2')
('', 2)
>>> parseExtn('sci')
('sci', 1)
```

stsci.tools.fileutil.**parseFilename**(*filename*)
Parse out filename from any specified extensions.

Returns rootname and string version of extension name.

stsci.tools.fileutil.**rAsciiLine**(*ifile*)
Returns the next non-blank line in an ASCII file.

stsci.tools.fileutil.**removeFile**(*inlist*)
Utility function for deleting a list of files or a single file.

This function will automatically delete both files of a GEIS image, just like 'iraf.imdelete'.

stsci.tools.fileutil.**reset**(**args*, ***kw*)
Set IRAF environment variables.

stsci.tools.fileutil.**set**(**args*, ***kw*)
Set IRAF environment variables.

stsci.tools.fileutil.**show**(**args*, ***kw*)
Print value of IRAF or OS environment variables.

stsci.tools.fileutil.**time**(***kw*)
Print current time and date.

stsci.tools.fileutil.**unset**(**args*, ***kw*)
Unset IRAF environment variables.

This is not a standard IRAF task, but it is obviously useful. It makes the resulting variables undefined. It silently ignores variables that are not defined. It does not change the os environment variables.

stsci.tools.fileutil.**untranslateName**(*s*)
Undo Python conversion of CL parameter or variable name.

stsci.tools.fileutil.**updateKeyword**(*filename*, *key*, *value*, *show=True*)
Add/update keyword to header with given value.

stsci.tools.fileutil.**verifyWriteMode**(*files*)
Checks whether files are writable. It is up to the calling routine to raise an Exception, if desired.

This function returns True, if all files are writable and False, if any are not writable. In addition, for all files found to not be writable, it will print out the list of names of affected files.

stsci.tools.parseinput.**checkASN**(*filename*)
Determine if the filename provided to the function belongs to an association.

Parameters
filename: string

Returns
validASN : boolean value

stsci.tools.parseinput.**countinputs**(*inputlist*)
Determine the number of inputfiles provided by the user and the number of those files that are association tables

Parameters
inputlist : string

the user input

Returns
numInputs: int

number of inputs provided by the user

numASNfiles: int

number of association files provided as input

stsci.tools.parseinput.**isValidAssocExtn**(*extname*)
Determine if the extension name given as input could represent a valid association file.

Parameters
extname : string

Returns
isValid : boolean value

stsci.tools.parseinput.**parseinput**(*inputlist*, *outputname=None*, *atfile=None*)
Recursively parse user input based upon the irafglob program and construct a list of files that need to be processed. This program addresses the following deficiencies of the irafglob program:

```
parseinput can extract filenames from association tables
```

Parameters
inputlist - string

specification of input files using either wild-cards, @-file or comma-separated list of filenames

outputname - string

desired name for output product to be created from the input files

atfile - object

function to use in interpreting the @-file columns that gets passed to irafglob

Returns

files - list of strings

names of output files to be processed

newoutputname - string

name of output file to be created.

See also:

stsci.tools.irafglob

License: http://www.stsci.edu/resources/software_hardware/pyraf/LICENSE

stsci.tools.irafglob.**irafglob**(*inlist*, *atfile=None*)

Returns a list of filenames based on the type of IRAF input.

Handles lists, wild-card characters, and at-files. For special at-files, use the atfile keyword to process them.

This function is recursive, so IRAF lists can also contain at-files and wild-card characters, e.g. *a.fits*, *@file.lst*, **flt.fits*.

1.1 STScI_Python Help Support

The *versioninfo* module reports the version information for a defined set of packages installed as part of STScI_Python which can be sent in as part of a help call. This information can then be used to help identify what software has been installed so that the source of the reported problem can be more easily identified.

stsci.tools.versioninfo.**printVersionInfo**()

CHAPTER

TWO

FITS/IMAGE UTILITIES

These modules provide support for working with FITS images, WCS information, or conversion of images to FITS format.

2.1 STPyFITS

The *stpyfits* module serves as a layer on top of PyFITS to support the use of single-valued arrays in extensions using the NPIX/PIXVALUE convention developed at STScI. The standard PyFITS module implements the strict FITS conventions, and these single-valued arrays are not part of the FITS standard. The stpyfits module is an extension to the `astropy.io.fits` module which offers additional features specific to STScI. These features include the handling of Constant Data Value Arrays.

exception `stsci.tools.stpyfits.`**`VerifyError`**
Bases: `exceptions.Exception`

Verify exception class.

class `stsci.tools.stpyfits.`**`Conf`**
Bases: `astropy.config.configuration.ConfigNamespace`

Configuration parameters for `astropy.io.fits`.

`enable_record_valued_keyword_cards`
If True, enable support for record-valued keywords as described by FITS WCS distortion paper. Otherwise they are treated as normal keywords.

`enable_uint`
If True, default to recognizing the convention for representing unsigned integers in FITS–if an array has BITPIX > 0, BSCALE = 1, and BZERO = 2**BITPIX, represent the data as unsigned integers per this convention.

`extension_name_case_sensitive`
If True, extension names (i.e. the `EXTNAME` keyword) should be treated as case-sensitive.

`strip_header_whitespace`
If True, automatically remove trailing whitespace for string values in headers. Otherwise the values are returned verbatim, with all whitespace intact.

`use_memmap`
If True, use memory-mapped file access to read/write the data in FITS files. This generally provides better performance, especially for large files, but may affect performance in I/O-heavy applications.

class `stsci.tools.stpyfits.`**`Card`**(*keyword=None*, *value=None*, *comment=None*, ***kwargs*)
Bases: `astropy.io.fits.verify._Verify`

ascardimage(**args*, ***kwargs*)
Deprecated since version 0.1: The ascardimage function is deprecated and may be removed in a future version. Use the *image* attribute instead.

classmethod fromstring(*image*)
Construct a *Card* object from a (raw) string. It will pad the string if it is not the length of a card image (80 columns). If the card image is longer than 80 columns, assume it contains CONTINUE card(s).

classmethod normalize_keyword(*keyword*)
classmethod to convert a keyword value that may contain a field-specifier to uppercase. The effect is to raise the key to uppercase and leave the field specifier in its original case.

Parameters
key : or str

A keyword value or a `keyword.field-specifier` value

cardimage
Deprecated since version 0.1: The cardimage function is deprecated and may be removed in a future version. Use the *image* attribute instead.

comment
Get the comment attribute from the card image if not already set.

field_specifier
The field-specifier of record-valued keyword cards; always *None* on normal cards.

image
The card "image", that is, the 80 byte character string that represents this card in an actual FITS header.

is_blank
True if the card is completely blank–that is, it has no keyword, value, or comment. It appears in the header as 80 spaces.

Returns *False* otherwise.

key
Deprecated since version 0.1: The key function is deprecated and may be removed in a future version. Use the *keyword* attribute instead.

keyword
Returns the keyword name parsed from the card image.

length = 80

rawkeyword
On record-valued keyword cards this is the name of the standard <= 8 character FITS keyword that this RVKC is stored in. Otherwise it is the card's normal keyword.

rawvalue
On record-valued keyword cards this is the raw string value in the `<field-specifier>: <value>` format stored in the card in order to represent a RVKC. Otherwise it is the card's normal value.

value
The value associated with the keyword stored in this card.

class stsci.tools.stpyfits.**CardList**(*cards=[]*, *keylist=None*)
Bases: list

Deprecated since version 0.1: *CardList* used to provide the list-like functionality for manipulating a header as a list of cards. This functionality is now subsumed into the *Header* class itself, so it is no longer necessary to create or use *CardList*s.

Construct the *CardList* object from a list of *Card* objects.

CardList is now merely a thin wrapper around *Header* to provide backwards compatibility for the old API. This should not be used for any new code.

Parameters
cards

A list of *Card* objects.

append(**args*, ***kwargs*)
Deprecated since version 0.1: The append function is deprecated and may be removed in a future version. Use *Header.append* instead.

Append a *Card* to the *CardList*.

Parameters
card : *Card* object

The *Card* to be appended.

useblanks : bool, optional

Use any *extra* blank cards?

If `useblanks` is *True*, and if there are blank cards directly before `END`, it will use this space first, instead of appending after these blank cards, so the total space will not increase. When `useblanks` is *False*, the card will be appended at the end, even if there are blank cards in front of `END`.

bottom : bool, optional

If *False* the card will be appended after the last non-commentary card. If *True* the card will be appended after the last non-blank card.

copy(**args*, ***kwargs*)
Deprecated since version 0.1: The copy function is deprecated and may be removed in a future version. Use *Header.copy* instead.

Make a (deep)copy of the *CardList*.

count(**args*, ***kwargs*)
Deprecated since version 0.1: The count function is deprecated and may be removed in a future version. Use *Header.count* instead.

count_blanks(**args*, ***kwargs*)
Deprecated since version 0.1: The count_blanks function is deprecated and may be removed in a future version.

Returns how many blank cards are *directly* before the `END` card.

extend(**args*, ***kwargs*)
Deprecated since version 0.1: The extend function is deprecated and may be removed in a future version. Use *Header.extend* instead.

filter_list(**args*, ***kwargs*)
Deprecated since version 0.1: The filter_list function is deprecated and may be removed in a future version. Use `header[<wildcard_pattern>]` instead.

Construct a *CardList* that contains references to all of the cards in this *CardList* that match the input key value including any special filter keys (`*`, `?`, and `...`).

Parameters
key : str

key value to filter the list with

Returns
cardlist

A *CardList* object containing references to all the requested cards.

index (**args*, ***kwargs*)
Deprecated since version 0.1: The index function is deprecated and may be removed in a future version. Use *Header.index* instead.

index_of (**args*, ***kwargs*)
Deprecated since version 0.1: The index_of function is deprecated and may be removed in a future version. Use *Header.index* instead.

Get the index of a keyword in the *CardList*.

Parameters
key : str or int

The keyword name (a string) or the index (an integer).

backward : bool, optional

When *True*, search the index from the `END`, i.e., backward.

Returns
index : int

The index of the *Card* with the given keyword.

insert (**args*, ***kwargs*)
Deprecated since version 0.1: The insert function is deprecated and may be removed in a future version. Use *Header.insert* instead.

Insert a *Card* to the *CardList*.

Parameters
pos : int

The position (index, keyword name will not be allowed) to insert. The new card will be inserted before it.

card : *Card* object

The card to be inserted.

useblanks : bool, optional

If `useblanks` is *True*, and if there are blank cards directly before `END`, it will use this space first, instead of appending after these blank cards, so the total space will not increase. When `useblanks` is *False*, the card will be appended at the end, even if there are blank cards in front of `END`.

keys (**args*, ***kwargs*)
Deprecated since version 0.1: The keys function is deprecated and may be removed in a future version. Use *Header.keys* instead.

Return a list of all keywords from the *CardList*.

pop (**args*, ***kwargs*)
Deprecated since version 0.1: The pop function is deprecated and may be removed in a future version. Use *Header.pop* instead.

remove(**args*, ***kwargs*)
Deprecated since version 0.1: The remove function is deprecated and may be removed in a future version. Use *Header.remove* instead.

values(**args*, ***kwargs*)
Deprecated since version 0.1: The values function is deprecated and may be removed in a future version. Use *Header.values* instead.

Return a list of the values of all cards in the *CardList*.

For RecordValuedKeywordCard objects, the value returned is the floating point value, exclusive of the field_specifier.

class stsci.tools.stpyfits.**Undefined**
Bases: object

Undefined value.

class stsci.tools.stpyfits.**Column**(*name=None*, *format=None*, *unit=None*, *null=None*, *bscale=None*, *bzero=None*, *disp=None*, *start=None*, *dim=None*, *array=None*, *ascii=None*)
Bases: astropy.io.fits.util.NotifierMixin

Class which contains the definition of one column, e.g. ttype, tform, etc. and the array containing values for the column.

Construct a *Column* by specifying attributes. All attributes except format can be optional; see column_creation and creating_ascii_table for more information regarding TFORM keyword.

Parameters

name : str, optional

column name, corresponding to TTYPE keyword

format : str

column format, corresponding to TFORM keyword

unit : str, optional

column unit, corresponding to TUNIT keyword

null : str, optional

null value, corresponding to TNULL keyword

bscale : int-like, optional

bscale value, corresponding to TSCAL keyword

bzero : int-like, optional

bzero value, corresponding to TZERO keyword

disp : str, optional

display format, corresponding to TDISP keyword

start : int, optional

column starting position (ASCII table only), corresponding to TBCOL keyword

dim : str, optional

column dimension corresponding to TDIM keyword

array : iterable, optional

a *list*, `numpy.ndarray` (or other iterable that can be used to initialize an ndarray) providing initial data for this column. The array will be automatically converted, if possible, to the data format of the column. In the case were non-trivial `bscale` and/or `bzero` arguments are given, the values in the array must be the *physical* values–that is, the values of column as if the scaling has already been applied (the array stored on the column object will then be converted back to its storage values).

ascii : bool, optional

set *True* if this describes a column for an ASCII table; this may be required to disambiguate the column format

copy()
Return a copy of this *Column*.

array
The Numpy `ndarray` associated with this *Column*.

If the column was instantiated with an array passed to the `array` argument, this will return that array. However, if the column is later added to a table, such as via `BinTableHDU.from_columns` as is typically the case, this attribute will be updated to reference the associated field in the table, which may no longer be the same array.

ascii
Whether this *Column* represents an column in an ASCII table.

bscale
Descriptor for attributes of *Column* that are associated with keywords in the FITS header and describe properties of the column as specified in the FITS standard.

Each *ColumnAttribute* may have a `validator` method defined on it. This validates values set on this attribute to ensure that they meet the FITS standard. Invalid values will raise a warning and will not be used in formatting the column. The validator should take two arguments–the *Column* it is being assigned to, and the new value for the attribute, and it must raise an *AssertionError* if the value is invalid.

The *ColumnAttribute* itself is a decorator that can be used to define the `validator` for each column attribute. For example:

```
@ColumnAttribute('TTYPE')
def name(col, name):
    assert isinstance(name, str)
```

The actual object returned by this decorator is the *ColumnAttribute* instance though, not the `name` function. As such `name` is not a method of the class it is defined in.

The setter for *ColumnAttribute* also updates the header of any table HDU this column is attached to in order to reflect the change. The `validator` should ensure that the value is valid for inclusion in a FITS header.

bzero
Descriptor for attributes of *Column* that are associated with keywords in the FITS header and describe properties of the column as specified in the FITS standard.

Each *ColumnAttribute* may have a `validator` method defined on it. This validates values set on this attribute to ensure that they meet the FITS standard. Invalid values will raise a warning and will not be used in formatting the column. The validator should take two arguments–the *Column* it is being assigned to, and the new value for the attribute, and it must raise an *AssertionError* if the value is invalid.

The *ColumnAttribute* itself is a decorator that can be used to define the `validator` for each column attribute. For example:

```
        @ColumnAttribute('TTYPE')
        def name(col, name):
            assert isinstance(name, str)
```

The actual object returned by this decorator is the *ColumnAttribute* instance though, not the `name` function. As such `name` is not a method of the class it is defined in.

The setter for *ColumnAttribute* also updates the header of any table HDU this column is attached to in order to reflect the change. The `validator` should ensure that the value is valid for inclusion in a FITS header.

dim

Descriptor for attributes of *Column* that are associated with keywords in the FITS header and describe properties of the column as specified in the FITS standard.

Each *ColumnAttribute* may have a `validator` method defined on it. This validates values set on this attribute to ensure that they meet the FITS standard. Invalid values will raise a warning and will not be used in formatting the column. The validator should take two arguments–the *Column* it is being assigned to, and the new value for the attribute, and it must raise an *AssertionError* if the value is invalid.

The *ColumnAttribute* itself is a decorator that can be used to define the `validator` for each column attribute. For example:

```
        @ColumnAttribute('TTYPE')
        def name(col, name):
            assert isinstance(name, str)
```

The actual object returned by this decorator is the *ColumnAttribute* instance though, not the `name` function. As such `name` is not a method of the class it is defined in.

The setter for *ColumnAttribute* also updates the header of any table HDU this column is attached to in order to reflect the change. The `validator` should ensure that the value is valid for inclusion in a FITS header.

disp

Descriptor for attributes of *Column* that are associated with keywords in the FITS header and describe properties of the column as specified in the FITS standard.

Each *ColumnAttribute* may have a `validator` method defined on it. This validates values set on this attribute to ensure that they meet the FITS standard. Invalid values will raise a warning and will not be used in formatting the column. The validator should take two arguments–the *Column* it is being assigned to, and the new value for the attribute, and it must raise an *AssertionError* if the value is invalid.

The *ColumnAttribute* itself is a decorator that can be used to define the `validator` for each column attribute. For example:

```
        @ColumnAttribute('TTYPE')
        def name(col, name):
            assert isinstance(name, str)
```

The actual object returned by this decorator is the *ColumnAttribute* instance though, not the `name` function. As such `name` is not a method of the class it is defined in.

The setter for *ColumnAttribute* also updates the header of any table HDU this column is attached to in order to reflect the change. The `validator` should ensure that the value is valid for inclusion in a FITS header.

dtype

format

Descriptor for attributes of *Column* that are associated with keywords in the FITS header and describe properties of the column as specified in the FITS standard.

Each *ColumnAttribute* may have a `validator` method defined on it. This validates values set on this attribute to ensure that they meet the FITS standard. Invalid values will raise a warning and will not be used in formatting the column. The validator should take two arguments–the *Column* it is being assigned to, and the new value for the attribute, and it must raise an *AssertionError* if the value is invalid.

The *ColumnAttribute* itself is a decorator that can be used to define the `validator` for each column attribute. For example:

```
@ColumnAttribute('TTYPE')
def name(col, name):
    assert isinstance(name, str)
```

The actual object returned by this decorator is the *ColumnAttribute* instance though, not the `name` function. As such `name` is not a method of the class it is defined in.

The setter for *ColumnAttribute* also updates the header of any table HDU this column is attached to in order to reflect the change. The `validator` should ensure that the value is valid for inclusion in a FITS header.

name

Descriptor for attributes of *Column* that are associated with keywords in the FITS header and describe properties of the column as specified in the FITS standard.

Each *ColumnAttribute* may have a `validator` method defined on it. This validates values set on this attribute to ensure that they meet the FITS standard. Invalid values will raise a warning and will not be used in formatting the column. The validator should take two arguments–the *Column* it is being assigned to, and the new value for the attribute, and it must raise an *AssertionError* if the value is invalid.

The *ColumnAttribute* itself is a decorator that can be used to define the `validator` for each column attribute. For example:

```
@ColumnAttribute('TTYPE')
def name(col, name):
    assert isinstance(name, str)
```

The actual object returned by this decorator is the *ColumnAttribute* instance though, not the `name` function. As such `name` is not a method of the class it is defined in.

The setter for *ColumnAttribute* also updates the header of any table HDU this column is attached to in order to reflect the change. The `validator` should ensure that the value is valid for inclusion in a FITS header.

null

Descriptor for attributes of *Column* that are associated with keywords in the FITS header and describe properties of the column as specified in the FITS standard.

Each *ColumnAttribute* may have a `validator` method defined on it. This validates values set on this attribute to ensure that they meet the FITS standard. Invalid values will raise a warning and will not be used in formatting the column. The validator should take two arguments–the *Column* it is being assigned to, and the new value for the attribute, and it must raise an *AssertionError* if the value is invalid.

The *ColumnAttribute* itself is a decorator that can be used to define the `validator` for each column attribute. For example:

```
@ColumnAttribute('TTYPE')
def name(col, name):
    assert isinstance(name, str)
```

The actual object returned by this decorator is the *ColumnAttribute* instance though, not the `name` function. As such `name` is not a method of the class it is defined in.

The setter for *ColumnAttribute* also updates the header of any table HDU this column is attached to in order to reflect the change. The `validator` should ensure that the value is valid for inclusion in a FITS header.

start

Descriptor for attributes of *Column* that are associated with keywords in the FITS header and describe properties of the column as specified in the FITS standard.

Each *ColumnAttribute* may have a `validator` method defined on it. This validates values set on this attribute to ensure that they meet the FITS standard. Invalid values will raise a warning and will not be used in formatting the column. The validator should take two arguments–the *Column* it is being assigned to, and the new value for the attribute, and it must raise an *AssertionError* if the value is invalid.

The *ColumnAttribute* itself is a decorator that can be used to define the `validator` for each column attribute. For example:

```
@ColumnAttribute('TTYPE')
def name(col, name):
    assert isinstance(name, str)
```

The actual object returned by this decorator is the *ColumnAttribute* instance though, not the `name` function. As such `name` is not a method of the class it is defined in.

The setter for *ColumnAttribute* also updates the header of any table HDU this column is attached to in order to reflect the change. The `validator` should ensure that the value is valid for inclusion in a FITS header.

unit

Descriptor for attributes of *Column* that are associated with keywords in the FITS header and describe properties of the column as specified in the FITS standard.

Each *ColumnAttribute* may have a `validator` method defined on it. This validates values set on this attribute to ensure that they meet the FITS standard. Invalid values will raise a warning and will not be used in formatting the column. The validator should take two arguments–the *Column* it is being assigned to, and the new value for the attribute, and it must raise an *AssertionError* if the value is invalid.

The *ColumnAttribute* itself is a decorator that can be used to define the `validator` for each column attribute. For example:

```
@ColumnAttribute('TTYPE')
def name(col, name):
    assert isinstance(name, str)
```

The actual object returned by this decorator is the *ColumnAttribute* instance though, not the `name` function. As such `name` is not a method of the class it is defined in.

The setter for *ColumnAttribute* also updates the header of any table HDU this column is attached to in order to reflect the change. The `validator` should ensure that the value is valid for inclusion in a FITS header.

class `stsci.tools.stpyfits.`**ColDefs**(*input*, *tbtype=None*, *ascii=False*)

Bases: `astropy.io.fits.util.NotifierMixin`

Column definitions class.

It has attributes corresponding to the *Column* attributes (e.g. *ColDefs* has the attribute `names` while *Column* has `name`). Each attribute in *ColDefs* is a list of corresponding attribute values from all *Column* objects.

Parameters

input : sequence of *Column*, *ColDefs*, other

An existing table HDU, an existing *ColDefs*, or any multi-field Numpy array or `numpy.recarray`.

****(Deprecated)** tbtype** : str, optional

which table HDU, `"BinTableHDU"` (default) or `"TableHDU"` (text table). Now ColDefs for a normal (binary) table by default, but converted automatically to ASCII table ColDefs in the appropriate contexts (namely, when creating an ASCII table).

ascii : bool

add_col(*column*)

Append one *Column* to the column definition.

change_attrib(*col_name*, *attrib*, *new_value*)

Change an attribute (in the `KEYWORD_ATTRIBUTES` list) of a *Column*.

Parameters

col_name : str or int

The column name or index to change

attrib : str

The attribute name

new_value : object

The new value for the attribute

change_name(*col_name*, *new_name*)

Change a *Column*'s name.

Parameters

col_name : str

The current name of the column

new_name : str

The new name of the column

change_unit(*col_name*, *new_unit*)

Change a *Column*'s unit.

Parameters

col_name : str or int

The column name or index

new_unit : str

The new unit for the column

del_col(*col_name*)

Delete (the definition of) one *Column*.

col_name

[str or int] The column's name or index

info(*attrib='all'*, *output=None*)

Get attribute(s) information of the column definition.

Parameters

attrib : str

Can be one or more of the attributes listed in `astropy.io.fits.column.KEYWORD_ATTRIBUTES`. The default is `"all"` which will print out all attributes. It forgives plurals and blanks. If there are two or more attribute names, they must be separated by comma(s).

output : file, optional

File-like object to output to. Outputs to stdout by default. If *False*, returns the attributes as a *dict* instead.

Notes

This function doesn't return anything by default; it just prints to stdout.

dtype

class `stsci.tools.stpyfits.`**Delayed**(*hdu=None*, *field=None*)

Bases: `object`

Delayed file-reading data.

class `stsci.tools.stpyfits.`**HDUList**(*hdus=[]*, *file=None*)

Bases: `list`, `astropy.io.fits.verify._Verify`

HDU list class. This is the top-level FITS object. When a FITS file is opened, a *HDUList* object is returned.

Construct a *HDUList* object.

Parameters

hdus : sequence of HDU objects or single HDU, optional

The HDU object(s) to comprise the *HDUList*. Should be instances of HDU classes like *ImageHDU* or *BinTableHDU*.

file : file object, optional

The opened physical file associated with the *HDUList*.

append(*hdu*)

Append a new HDU to the *HDUList*.

Parameters

hdu : HDU object

HDU to add to the *HDUList*.

close(*output_verify='exception'*, *verbose=False*, *closed=True*)

Close the associated FITS file and memmap object, if any.

Parameters

output_verify : str

Output verification option. Must be one of `"fix"`, `"silentfix"`, `"ignore"`, `"warn"`, or `"exception"`. May also be any combination of `"fix"` or `"silentfix"` with `"+ignore"`, `+warn`, or `+exception"` (e.g. ``"fix+warn"`). See verify for more info.

verbose : bool

When *True*, print out verbose messages.

closed : bool

When *True*, close the underlying file object.

fileinfo(*index*)

Returns a dictionary detailing information about the locations of the indexed HDU within any associated file. The values are only valid after a read or write of the associated file with no intervening changes to the *HDUList*.

Parameters

index : int

Index of HDU for which info is to be returned.

Returns

fileinfo : dict or None

The dictionary details information about the locations of the indexed HDU within an associated file. Returns *None* when the HDU is not associated with a file.

Dictionary contents:

Key	Value
file	File object associated with the HDU
filename	Name of associated file object
filemode	Mode in which the file was opened (readonly, update, append, denywrite, ostream)
resized	Flag that when *True* indicates that the data has been resized since the last read/write so the returned values may not be valid.
hdrLoc	Starting byte location of header in file
datLoc	Starting byte location of data block in file
datSpan	Data size including padding

filename()

Return the file name associated with the HDUList object if one exists. Otherwise returns None.

Returns

filename : a string containing the file name associated with the

HDUList object if an association exists. Otherwise returns None.

flush(*output_verify='fix'*, *verbose=False*)

Force a write of the *HDUList* back to the file (for append and update modes only).

Parameters

output_verify : str

Output verification option. Must be one of `"fix"`, `"silentfix"`, `"ignore"`, `"warn"`, or `"exception"`. May also be any combination of `"fix"` or `"silentfix"` with `"+ignore"`, `+warn`, or `+exception"` (e.g. ``"fix+warn"`). See verify for more info.

verbose : bool

When *True*, print verbose messages

classmethod fromfile(*fileobj*, *mode=None*, *memmap=None*, *save_backup=False*, *cache=True*, ***kwargs*)

Creates an *HDUList* instance from a file-like object.

The actual implementation of `fitsopen()`, and generally shouldn't be used directly. Use *open* instead (and see its documentation for details of the parameters accepted by this method).

classmethod fromstring(*data*, ***kwargs*)

Creates an *HDUList* instance from a string or other in-memory data buffer containing an entire FITS file. Similar to *HDUList.fromfile*, but does not accept the mode or memmap arguments, as they are only relevant to reading from a file on disk.

This is useful for interfacing with other libraries such as CFITSIO, and may also be useful for streaming applications.

Parameters

data : str, buffer, memoryview, etc.

A string or other memory buffer containing an entire FITS file. It should be noted that if that memory is read-only (such as a Python string) the returned *HDUList*'s data portions will also be read-only.

kwargs : dict

Optional keyword arguments. See `astropy.io.fits.open` for details.

Returns

hdul : HDUList

An *HDUList* object representing the in-memory FITS file.

index_of(*key*)

Get the index of an HDU from the *HDUList*.

Parameters

key : int, str or tuple of (string, int)

The key identifying the HDU. If `key` is a tuple, it is of the form `(key, ver)` where `ver` is an `EXTVER` value that must match the HDU being searched for.

Returns

index : int

The index of the HDU in the *HDUList*.

info(*output=None*)

Summarize the info of the HDUs in this *HDUList*.

Note that this function prints its results to the console—it does not return a value.

Parameters

output : file, bool, optional

A file-like object to write the output to. If *False*, does not output to a file and instead returns a list of tuples representing the HDU info. Writes to `sys.stdout` by default.

insert(*index*, *hdu*)

Insert an HDU into the *HDUList* at the given `index`.

Parameters

index : int

Index before which to insert the new HDU.

hdu : HDU object

The HDU object to insert

readall()
Read data of all HDUs into memory.

update_extend()
Make sure that if the primary header needs the keyword EXTEND that it has it and it is correct.

writeto(*fileobj*, *output_verify='exception'*, *clobber=False*, *checksum=False*)
Write the *HDUList* to a new file.

Parameters
fileobj : file path, file object or file-like object

File to write to. If a file object, must be opened in a writeable mode.

output_verify : str

Output verification option. Must be one of "fix", "silentfix", "ignore", "warn", or "exception". May also be any combination of "fix" or "silentfix" with "+ignore", +warn, or +exception" (e.g. ``"fix+warn"). See verify for more info.

clobber : bool

When *True*, overwrite the output file if exists.

checksum : bool

When *True* adds both DATASUM and CHECKSUM cards to the headers of all HDU's written to the file.

stsci.tools.stpyfits.**PrimaryHDU**
alias of *ConstantValuePrimaryHDU*

stsci.tools.stpyfits.**ImageHDU**
alias of *ConstantValueImageHDU*

class stsci.tools.stpyfits.**TableHDU**(*data=None*, *header=None*, *name=None*)
Bases: astropy.io.fits.hdu.table._TableBaseHDU

FITS ASCII table extension HDU class.

classmethod match_header(*header*)

class stsci.tools.stpyfits.**BinTableHDU**(*data=None*, *header=None*, *name=None*, *uint=False*)
Bases: astropy.io.fits.hdu.table._TableBaseHDU

Binary table HDU class.

Parameters
header : Header instance

header to be used

data : array

data to be used

name : str

name to be populated in EXTNAME keyword

uint : bool, optional

set to *True* if the table contains unsigned integer columns.

dump(*datafile=None*, *cdfile=None*, *hfile=None*, *clobber=False*)

Dump the table HDU to a file in ASCII format. The table may be dumped in three separate files, one containing column definitions, one containing header parameters, and one for table data.

Parameters

datafile : file path, file object or file-like object, optional

Output data file. The default is the root name of the fits file associated with this HDU appended with the extension `.txt`.

cdfile : file path, file object or file-like object, optional

Output column definitions file. The default is *None*, no column definitions output is produced.

hfile : file path, file object or file-like object, optional

Output header parameters file. The default is *None*, no header parameters output is produced.

clobber : bool

Overwrite the output files if they exist.

Notes

The primary use for the *dump* method is to allow viewing and editing the table data and parameters in a standard text editor. The *load* method can be used to create a new table from the three plain text (ASCII) files.

- **datafile:** Each line of the data file represents one row of table data. The data is output one column at a time in column order. If a column contains an array, each element of the column array in the current row is output before moving on to the next column. Each row ends with a new line.

  Integer data is output right-justified in a 21-character field followed by a blank. Floating point data is output right justified using 'g' format in a 21-character field with 15 digits of precision, followed by a blank. String data that does not contain whitespace is output left-justified in a field whose width matches the width specified in the `TFORM` header parameter for the column, followed by a blank. When the string data contains whitespace characters, the string is enclosed in quotation marks (`""`). For the last data element in a row, the trailing blank in the field is replaced by a new line character.

  For column data containing variable length arrays ('P' format), the array data is preceded by the string `'VLA_Length= '` and the integer length of the array for that row, left-justified in a 21-character field, followed by a blank.

  Note: This format does *not* support variable length arrays using the ('Q' format) due to difficult to overcome ambiguities. What this means is that this file format cannot support VLA columns in tables stored in files that are over 2 GB in size.

  For column data representing a bit field ('X' format), each bit value in the field is output right-justified in a 21-character field as 1 (for true) or 0 (for false).

- **cdfile:** Each line of the column definitions file provides the definitions for one column in the table. The line is broken up into 8, sixteen-character fields. The first field provides the column name (`TTYPEn`). The second field provides the column format (`TFORMn`). The third field provides the display format (`TDISPn`). The fourth field provides the physical units (`TUNITn`). The fifth field provides the dimensions for a multidimensional array (`TDIMn`). The sixth field provides the value that signifies an undefined value (`TNULLn`). The seventh field provides the scale factor (`TSCALn`). The eighth field

provides the offset value (TZEROn). A field value of "" is used to represent the case where no value is provided.

- **hfile:** Each line of the header parameters file provides the definition of a single HDU header card as represented by the card image.

classmethod load(*datafile*, *cdfile=None*, *hfile=None*, *replace=False*, *header=None*)

Create a table from the input ASCII files. The input is from up to three separate files, one containing column definitions, one containing header parameters, and one containing column data.

The column definition and header parameters files are not required. When absent the column definitions and/or header parameters are taken from the header object given in the header argument; otherwise sensible defaults are inferred (though this mode is not recommended).

Parameters

datafile : file path, file object or file-like object

Input data file containing the table data in ASCII format.

cdfile : file path, file object, file-like object, optional

Input column definition file containing the names, formats, display formats, physical units, multidimensional array dimensions, undefined values, scale factors, and offsets associated with the columns in the table. If *None*, the column definitions are taken from the current values in this object.

hfile : file path, file object, file-like object, optional

Input parameter definition file containing the header parameter definitions to be associated with the table. If *None*, the header parameter definitions are taken from the current values in this objects header.

replace : bool

When *True*, indicates that the entire header should be replaced with the contents of the ASCII file instead of just updating the current header.

header : Header object

When the cdfile and hfile are missing, use this Header object in the creation of the new table and HDU. Otherwise this Header supercedes the keywords from hfile, which is only used to update values not present in this Header, unless `replace=True` in which this Header's values are completely replaced with the values from hfile.

Notes

The primary use for the *load* method is to allow the input of ASCII data that was edited in a standard text editor of the table data and parameters. The *dump* method can be used to create the initial ASCII files.

- **datafile:** Each line of the data file represents one row of table data. The data is output one column at a time in column order. If a column contains an array, each element of the column array in the current row is output before moving on to the next column. Each row ends with a new line.

  Integer data is output right-justified in a 21-character field followed by a blank. Floating point data is output right justified using 'g' format in a 21-character field with 15 digits of precision, followed by a blank. String data that does not contain whitespace is output left-justified in a field whose width matches the width specified in the TFORM header parameter for the column, followed by a blank. When the string data contains whitespace characters, the string is enclosed in quotation marks (""). For the last data element in a row, the trailing blank in the field is replaced by a new line character.

  For column data containing variable length arrays ('P' format), the array data is preceded by the string 'VLA_Length= ' and the integer length of the array for that row, left-justified in a 21-character

field, followed by a blank.

Note: This format does *not* support variable length arrays using the ('Q' format) due to difficult to overcome ambiguities. What this means is that this file format cannot support VLA columns in tables stored in files that are over 2 GB in size.

For column data representing a bit field ('X' format), each bit value in the field is output right-justified in a 21-character field as 1 (for true) or 0 (for false).

- **cdfile:** Each line of the column definitions file provides the definitions for one column in the table. The line is broken up into 8, sixteen-character fields. The first field provides the column name (`TTYPEn`). The second field provides the column format (`TFORMn`). The third field provides the display format (`TDISPn`). The fourth field provides the physical units (`TUNITn`). The fifth field provides the dimensions for a multidimensional array (`TDIMn`). The sixth field provides the value that signifies an undefined value (`TNULLn`). The seventh field provides the scale factor (`TSCALn`). The eighth field provides the offset value (`TZEROn`). A field value of `""` is used to represent the case where no value is provided.
- **hfile:** Each line of the header parameters file provides the definition of a single HDU header card as represented by the card image.

classmethod match_header(*header*)

classmethod tcreate(**args*, ***kwargs*)

Deprecated since version 0.1: The tcreate method is deprecated and may be removed in a future version. Use *load* instead.

tdump(**args*, ***kwargs*)

Deprecated since version 0.1: The tdump function is deprecated and may be removed in a future version. Use *dump* instead.

class stsci.tools.stpyfits.**GroupsHDU**(*data=None*, *header=None*)

Bases: astropy.io.fits.hdu.image.PrimaryHDU, astropy.io.fits.hdu.table._TableLikeHDU

FITS Random Groups HDU class.

See the random-groups section in the PyFITS documentation for more details on working with this type of HDU.

classmethod match_header(*header*)

update_header()

columns

data

The data of a random group FITS file will be like a binary table's data.

is_image

parnames

The names of the group parameters as described by the header.

size

Returns the size (in bytes) of the HDU's data part.

class stsci.tools.stpyfits.**GroupData**
Bases: astropy.io.fits.fitsrec.FITS_rec

Random groups data object.

Allows structured access to FITS Group data in a manner analogous to tables.

par(*parname*)
Get the group parameter values.

data
The raw group data represented as a multi-dimensional numpy.ndarray array.

class stsci.tools.stpyfits.**Group**(*input*, *row=0*, *start=None*, *end=None*, *step=None*, *base=None*)
Bases: astropy.io.fits.fitsrec.FITS_record

One group of the random group data.

par(*parname*)
Get the group parameter value.

setpar(*parname*, *value*)
Set the group parameter value.

data

parnames

class stsci.tools.stpyfits.**CompImageHDU**(*data=None*, *header=None*, *name=None*, *compression_type='RICE_1'*, *tile_size=None*, *hcomp_scale=0*, *hcomp_smooth=0*, *quantize_level=16.0*, *quantize_method=-1*, *dither_seed=0*, *do_not_scale_image_data=False*, *uint=False*, *scale_back=False*, ***kwargs*)
Bases: astropy.io.fits.hdu.table.BinTableHDU

Compressed Image HDU class.

Parameters

data : array, optional

Uncompressed image data

header : Header instance, optional

Header to be associated with the image; when reading the HDU from a file (data=DELAYED), the header read from the file

name : str, optional

The EXTNAME value; if this value is *None*, then the name from the input image header will be used; if there is no name in the input image header then the default name COMPRESSED_IMAGE is used.

compression_type : str, optional

Compression algorithm: one of 'RICE_1','RICE_ONE','PLIO_1','GZIP_1', 'GZIP_2','HCOMPRESS_1'

tile_size : int, optional

Compression tile sizes. Default treats each row of image as a tile.

hcomp_scale : float, optional

HCOMPRESS scale parameter

hcomp_smooth : float, optional

HCOMPRESS smooth parameter

quantize_level : float, optional

Floating point quantization level; see note below

quantize_method : int, optional

Floating point quantization dithering method; can be either `NO_DITHER` (-1), `SUBTRACTIVE_DITHER_1` (1; default), or `SUBTRACTIVE_DITHER_2` (2); see note below

dither_seed : int, optional

Random seed to use for dithering; can be either an integer in the range 1 to 1000 (inclusive), `DITHER_SEED_CLOCK` (0; default), or `DITHER_SEED_CHECKSUM` (-1); see note below

Notes

The astropy.io.fits package supports 2 methods of image compression:

1. The entire FITS file may be externally compressed with the gzip or pkzip utility programs, producing a `*.gz` or `*.zip` file, respectively. When reading compressed files of this type, Astropy first uncompresses the entire file into a temporary file before performing the requested read operations. The astropy.io.fits package does not support writing to these types of compressed files. This type of compression is supported in the `_File` class, not in the *`CompImageHDU`* class. The file compression type is recognized by the `.gz` or `.zip` file name extension.

2. The *`CompImageHDU`* class supports the FITS tiled image compression convention in which the image is subdivided into a grid of rectangular tiles, and each tile of pixels is individually compressed. The details of this FITS compression convention are described at the FITS Support Office web site. Basically, the compressed image tiles are stored in rows of a variable length array column in a FITS binary table. The astropy.io.fits recognizes that this binary table extension contains an image and treats it as if it were an image extension. Under this tile-compression format, FITS header keywords remain uncompressed. At this time, Astropy does not support the ability to extract and uncompress sections of the image without having to uncompress the entire image.

The astropy.io.fits package supports 3 general-purpose compression algorithms plus one other special-purpose compression technique that is designed for data masks with positive integer pixel values. The 3 general purpose algorithms are GZIP, Rice, and HCOMPRESS, and the special-purpose technique is the IRAF pixel list compression technique (PLIO). The `compression_type` parameter defines the compression algorithm to be used.

The FITS image can be subdivided into any desired rectangular grid of compression tiles. With the GZIP, Rice, and PLIO algorithms, the default is to take each row of the image as a tile. The HCOMPRESS algorithm is inherently 2-dimensional in nature, so the default in this case is to take 16 rows of the image per tile. In most cases, it makes little difference what tiling pattern is used, so the default tiles are usually adequate. In the case of very small images, it could be more efficient to compress the whole image as a single tile. Note that the image dimensions are not required to be an integer multiple of the tile dimensions; if not, then the tiles at the edges of the image will be smaller than the other tiles. The `tile_size` parameter may be provided as a list of tile sizes, one for each dimension in the image. For example a `tile_size` value of `[100,100]` would divide a 300 X 300 image into 9 100 X 100 tiles.

The 4 supported image compression algorithms are all 'lossless' when applied to integer FITS images; the pixel values are preserved exactly with no loss of information during the compression and uncompression process. In addition, the HCOMPRESS algorithm supports a 'lossy' compression mode that will produce larger amount of

image compression. This is achieved by specifying a non-zero value for the `hcomp_scale` parameter. Since the amount of compression that is achieved depends directly on the RMS noise in the image, it is usually more convenient to specify the `hcomp_scale` factor relative to the RMS noise. Setting `hcomp_scale = 2.5` means use a scale factor that is 2.5 times the calculated RMS noise in the image tile. In some cases it may be desirable to specify the exact scaling to be used, instead of specifying it relative to the calculated noise value. This may be done by specifying the negative of the desired scale value (typically in the range -2 to -100).

Very high compression factors (of 100 or more) can be achieved by using large `hcomp_scale` values, however, this can produce undesirable 'blocky' artifacts in the compressed image. A variation of the HCOMPRESS algorithm (called HSCOMPRESS) can be used in this case to apply a small amount of smoothing of the image when it is uncompressed to help cover up these artifacts. This smoothing is purely cosmetic and does not cause any significant change to the image pixel values. Setting the `hcomp_smooth` parameter to 1 will engage the smoothing algorithm.

Floating point FITS images (which have `BITPIX` = -32 or -64) usually contain too much 'noise' in the least significant bits of the mantissa of the pixel values to be effectively compressed with any lossless algorithm. Consequently, floating point images are first quantized into scaled integer pixel values (and thus throwing away much of the noise) before being compressed with the specified algorithm (either GZIP, RICE, or HCOMPRESS). This technique produces much higher compression factors than simply using the GZIP utility to externally compress the whole FITS file, but it also means that the original floating point value pixel values are not exactly preserved. When done properly, this integer scaling technique will only discard the insignificant noise while still preserving all the real information in the image. The amount of precision that is retained in the pixel values is controlled by the `quantize_level` parameter. Larger values will result in compressed images whose pixels more closely match the floating point pixel values, but at the same time the amount of compression that is achieved will be reduced. Users should experiment with different values for this parameter to determine the optimal value that preserves all the useful information in the image, without needlessly preserving all the 'noise' which will hurt the compression efficiency.

The default value for the `quantize_level` scale factor is 16, which means that scaled integer pixel values will be quantized such that the difference between adjacent integer values will be 1/16th of the noise level in the image background. An optimized algorithm is used to accurately estimate the noise in the image. As an example, if the RMS noise in the background pixels of an image = 32.0, then the spacing between adjacent scaled integer pixel values will equal 2.0 by default. Note that the RMS noise is independently calculated for each tile of the image, so the resulting integer scaling factor may fluctuate slightly for each tile. In some cases, it may be desirable to specify the exact quantization level to be used, instead of specifying it relative to the calculated noise value. This may be done by specifying the negative of desired quantization level for the value of `quantize_level`. In the previous example, one could specify `quantize_level = -2.0` so that the quantized integer levels differ by 2.0. Larger negative values for `quantize_level` means that the levels are more coarsely-spaced, and will produce higher compression factors.

The quantization algorithm can also apply one of two random dithering methods in order to reduce bias in the measured intensity of background regions. The default method, specified with the constant `SUBTRACTIVE_DITHER_1` adds dithering to the zero-point of the quantization array itself rather than adding noise to the actual image. The random noise is added on a pixel-by-pixel basis, so in order restore each pixel from its integer value to its floating point value it is necessary to replay the same sequence of random numbers for each pixel (see below). The other method, `SUBTRACTIVE_DITHER_2`, is exactly like the first except that before dithering any pixel with a floating point value of `0.0` is replaced with the special integer value `-2147483647`. When the image is uncompressed, pixels with this value are restored back to `0.0` exactly. Finally, a value of `NO_DITHER` disables dithering entirely.

As mentioned above, when using the subtractive dithering algorithm it is necessary to be able to generate a (pseudo-)random sequence of noise for each pixel, and replay that same sequence upon decompressing. To facilitate this, a random seed between 1 and 10000 (inclusive) is used to seed a random number generator, and that seed is stored in the `ZDITHER0` keyword in the header of the compressed HDU. In order to use that seed to generate the same sequence of random numbers the same random number generator must be used at compression and decompression time; for that reason the tiled image convention provides an implementation

of a very simple pseudo-random number generator. The seed itself can be provided in one of three ways, controllable by the `dither_seed` argument: It may be specified manually, or it may be generated arbitrarily based on the system's clock (`DITHER_SEED_CLOCK`) or based on a checksum of the pixels in the image's first tile (`DITHER_SEED_CHECKSUM`). The clock-based method is the default, and is sufficient to ensure that the value is reasonably "arbitrary" and that the same seed is unlikely to be generated sequentially. The checksum method, on the other hand, ensures that the same seed is used every time for a specific image. This is particularly useful for software testing as it ensures that the same image will always use the same seed.

classmethod match_header(*header*)

scale(*type=None*, *option='old'*, *bscale=1*, *bzero=0*)

Scale image data by using `BSCALE` and `BZERO`.

Calling this method will scale `self.data` and update the keywords of `BSCALE` and `BZERO` in `self._header` and `self._image_header`. This method should only be used right before writing to the output file, as the data will be scaled and is therefore not very usable after the call.

Parameters

type : str, optional

destination data type, use a string representing a numpy dtype name, (e.g. `'uint8'`, `'int16'`, `'float32'` etc.). If is *None*, use the current data type.

option : str, optional

how to scale the data: if `"old"`, use the original `BSCALE` and `BZERO` values when the data was read/created. If `"minmax"`, use the minimum and maximum of the data to scale. The option will be overwritten by any user-specified bscale/bzero values.

bscale, bzero : int, optional

user specified `BSCALE` and `BZERO` values.

updateCompressedData(**args*, ***kwargs*)

Deprecated since version 0.3: The updateCompressedData function is deprecated and may be removed in a future version. Use (refactor your code) instead.

updateHeader(**args*, ***kwargs*)

Deprecated since version 0.3: The updateHeader function is deprecated and may be removed in a future version. Use (refactor your code; this function no longer does anything) instead.

updateHeaderData(**args*, ***kwargs*)

Deprecated since version 0.3: The updateHeaderData function will be deprecated in a future version. Use (refactor your code) instead.

DEPRECATED_KWARGS = {'quantizeLevel': 'quantize_level', 'compressionType': 'compression_type', 'hcompScale': 'hc

compData

Deprecated since version 0.3: The compData function will be deprecated in a future version. Use the `compressed_data` attribute instead.

compressed_data

data

header

shape
Shape of the image array–should be equivalent to `self.data.shape`.

class `stsci.tools.stpyfits.`**FitsHDU**(*data=None*, *header=None*, *name=None*, ***kwargs*)
Bases: `astropy.io.fits.hdu.base.NonstandardExtHDU`

A non-standard extension HDU for encapsulating entire FITS files within a single HDU of a container FITS file. These HDUs have an extension (that is an XTENSION keyword) of FITS.

The FITS file contained in the HDU's data can be accessed by the *hdulist* attribute which returns the contained FITS file as an *HDUList* object.

classmethod fromfile(*filename*, *compress=False*)
Like *FitsHDU.fromhdulist()*, but creates a FitsHDU from a file on disk.

Parameters
filename : str

The path to the file to read into a FitsHDU

compress : bool, optional

Gzip compress the FITS file

classmethod fromhdulist(*hdulist*, *compress=False*)
Creates a new FitsHDU from a given HDUList object.

Parameters
hdulist : HDUList

A valid Headerlet object.

compress : bool, optional

Gzip compress the FITS file

classmethod match_header(*header*)

hdulist

class `stsci.tools.stpyfits.`**StreamingHDU**(*name*, *header*)
Bases: `object`

A class that provides the capability to stream data to a FITS file instead of requiring data to all be written at once.

The following pseudocode illustrates its use:

```
header = astropy.io.fits.Header()

for all the cards you need in the header:
    header[key] = (value, comment)

shdu = astropy.io.fits.StreamingHDU('filename.fits', header)

for each piece of data:
    shdu.write(data)

shdu.close()
```

Construct a *StreamingHDU* object given a file name and a header.

Parameters

name : file path, file object, or file like object

The file to which the header and data will be streamed. If opened, the file object must be opened in a writeable binary mode such as 'wb' or 'ab+'.

header : *Header* instance

The header object associated with the data to be written to the file.

Notes

The file will be opened and the header appended to the end of the file. If the file does not already exist, it will be created, and if the header represents a Primary header, it will be written to the beginning of the file. If the file does not exist and the provided header is not a Primary header, a default Primary HDU will be inserted at the beginning of the file and the provided header will be added as the first extension. If the file does already exist, but the provided header represents a Primary header, the header will be modified to an image extension header and appended to the end of the file.

close()

Close the physical FITS file.

write(*data*)

Write the given data to the stream.

Parameters

data : ndarray

Data to stream to the file.

Returns

writecomplete : int

Flag that when *True* indicates that all of the required data has been written to the stream.

Notes

Only the amount of data specified in the header provided to the class constructor may be written to the stream. If the provided data would cause the stream to overflow, an *IOError* exception is raised and the data is not written. Once sufficient data has been written to the stream to satisfy the amount specified in the header, the stream is padded to fill a complete FITS block and no more data will be accepted. An attempt to write more data after the stream has been filled will raise an *IOError* exception. If the dtype of the input data does not match what is expected by the header, a *TypeError* exception is raised.

size

Return the size (in bytes) of the data portion of the HDU.

class stsci.tools.stpyfits.**FITS_record**(*input*, *row=0*, *start=None*, *end=None*, *step=None*, *base=None*, ***kwargs*)

Bases: object

FITS record class.

FITS_record is used to access records of the *FITS_rec* object. This will allow us to deal with scaled columns. It also handles conversion/scaling of columns in ASCII tables. The *FITS_record* class expects a *FITS_rec* object as input.

Parameters

input : array

The array to wrap.

row : int, optional

The starting logical row of the array.

start : int, optional

The starting column in the row associated with this object. Used for subsetting the columns of the *FITS_rec* object.

end : int, optional

The ending column in the row associated with this object. Used for subsetting the columns of the *FITS_rec* object.

field(*field*)
Get the field data of the record.

setfield(*field*, *value*)
Set the field data of the record.

class stsci.tools.stpyfits.**FITS_rec**
Bases: numpy.core.records.recarray

FITS record array class.

FITS_rec is the data part of a table HDU's data part. This is a layer over the recarray, so we can deal with scaled columns.

It inherits all of the standard methods from numpy.ndarray.

copy(*order='C'*)
The Numpy documentation lies; numpy.ndarray.copy is not equivalent to numpy.copy. Differences include that it re-views the copied array as self's ndarray subclass, as though it were taking a slice; this means __array_finalize__ is called and the copy shares all the array attributes (including ._converted!). So we need to make a deep copy of all those attributes so that the two arrays truly do not share any data.

field(*key*)
A view of a *Column*'s data as an array.

classmethod from_columns(*columns*, *nrows=0*, *fill=False*)
Given a *ColDefs* object of unknown origin, initialize a new *FITS_rec* object.

Note: This was originally part of the *new_table* function in the table module but was moved into a class method since most of its functionality always had more to do with initializing a *FITS_rec* object than anything else, and much of it also overlapped with FITS_rec._scale_back.

Parameters

columns : sequence of *Column* or a *ColDefs*

The columns from which to create the table data. If these columns have data arrays attached that data may be used in initializing the new table. Otherwise the input columns will be used as a template for a new table with the requested number of rows.

nrows : int

Number of rows in the new table. If the input columns have data associated with them, the size of the largest input column is used. Otherwise the default is 0.

fill : bool

If *True*, will fill all cells with zeros or blanks. If *False*, copy the data from input, undefined cells will still be filled with zeros/blanks.

columns
A user-visible accessor for the coldefs.

See https://aeon.stsci.edu/ssb/trac/pyfits/ticket/44

formats
List of column FITS formats.

names
List of column names.

class stsci.tools.stpyfits.**GroupData**
Bases: astropy.io.fits.fitsrec.FITS_rec

Random groups data object.

Allows structured access to FITS Group data in a manner analogous to tables.

par(*parname*)
Get the group parameter values.

data
The raw group data represented as a multi-dimensional numpy.ndarray array.

class stsci.tools.stpyfits.**Section**(*hdu*)
Bases: object

Image section.

Slices of this object load the corresponding section of an image array from the underlying FITS file on disk, and applies any BSCALE/BZERO factors.

Section slices cannot be assigned to, and modifications to a section are not saved back to the underlying file.

See the data-sections section of the PyFITS documentation for more details.

class stsci.tools.stpyfits.**Header**(*cards=[]*, *txtfile=None*)
Bases: object

FITS header class. This class exposes both a dict-like interface and a list-like interface to FITS headers.

The header may be indexed by keyword and, like a dict, the associated value will be returned. When the header contains cards with duplicate keywords, only the value of the first card with the given keyword will be returned. It is also possible to use a 2-tuple as the index in the form (keyword, n)–this returns the n-th value with that keyword, in the case where there are duplicate keywords.

For example:

```
>>> header['NAXIS']
0
>>> header[('FOO', 1)]  # Return the value of the second FOO keyword
'foo'
```

The header may also be indexed by card number:

```
>>> header[0]  # Return the value of the first card in the header
'T'
```

Commentary keywords such as HISTORY and COMMENT are special cases: When indexing the Header object with either 'HISTORY' or 'COMMENT' a list of all the HISTORY/COMMENT values is returned:

```
>>> header['HISTORY']
This is the first history entry in this header.
This is the second history entry in this header.
...
```

See the Astropy documentation for more details on working with headers.

Construct a *Header* from an iterable and/or text file.

Parameters

cards : A list of *Card* objects, optional

The cards to initialize the header with. Also allowed are other *Header* (or *dict*-like) objects.

txtfile : file path, file object or file-like object, optional

Input ASCII header parameters file **(Deprecated)** Use the Header.fromfile classmethod instead.

add_blank(*value=''*, *before=None*, *after=None*)

Add a blank card.

Parameters

value : str, optional

Text to be added.

before : str or int, optional

Same as in *Header.update*

after : str or int, optional

Same as in *Header.update*

add_comment(*value*, *before=None*, *after=None*)

Add a COMMENT card.

Parameters

value : str

Text to be added.

before : str or int, optional

Same as in *Header.update*

after : str or int, optional

Same as in *Header.update*

add_history(*value*, *before=None*, *after=None*)

Add a HISTORY card.

Parameters

value : str

History text to be added.

before : str or int, optional

Same as in *Header.update*

after : str or int, optional

Same as in *Header.update*

append(*card=None*, *useblanks=True*, *bottom=False*, *end=False*)

Appends a new keyword+value card to the end of the Header, similar to *list.append*.

By default if the last cards in the Header have commentary keywords, this will append the new keyword before the commentary (unless the new keyword is also commentary).

Also differs from *list.append* in that it can be called with no arguments: In this case a blank card is appended to the end of the Header. In the case all the keyword arguments are ignored.

Parameters

card : str, tuple

A keyword or a (keyword, value, [comment]) tuple representing a single header card; the comment is optional in which case a 2-tuple may be used

useblanks : bool, optional

If there are blank cards at the end of the Header, replace the first blank card so that the total number of cards in the Header does not increase. Otherwise preserve the number of blank cards.

bottom : bool, optional

If True, instead of appending after the last non-commentary card, append after the last non-blank card.

end : bool, optional

If True, ignore the useblanks and bottom options, and append at the very end of the Header.

clear()

Remove all cards from the header.

copy(*strip=False*)

Make a copy of the *Header*.

Parameters

strip : bool, optional

If *True*, strip any headers that are specific to one of the standard HDU types, so that this header can be used in a different HDU.

Returns

header

A new *Header* instance.

count(*keyword*)

Returns the count of the given keyword in the header, similar to *list.count* if the Header object is treated as a list of keywords.

Parameters

keyword : str

The keyword to count instances of in the header

extend(*cards*, *strip=True*, *unique=False*, *update=False*, *update_first=False*, *useblanks=True*, *bottom=False*, *end=False*)

Appends multiple keyword+value cards to the end of the header, similar to *list.extend*.

Parameters

cards : iterable

An iterable of (keyword, value, [comment]) tuples; see *Header.append*.

strip : bool, optional

Remove any keywords that have meaning only to specific types of HDUs, so that only more general keywords are added from extension Header or Card list (default: *True*).

unique : bool, optional

If *True*, ensures that no duplicate keywords are appended; keywords already in this header are simply discarded. The exception is commentary keywords (COMMENT, HISTORY, etc.): they are only treated as duplicates if their values match.

update : bool, optional

If *True*, update the current header with the values and comments from duplicate keywords in the input header. This supercedes the `unique` argument. Commentary keywords are treated the same as if `unique=True`.

update_first : bool, optional

If the first keyword in the header is 'SIMPLE', and the first keyword in the input header is 'XTENSION', the 'SIMPLE' keyword is replaced by the 'XTENSION' keyword. Likewise if the first keyword in the header is 'XTENSION' and the first keyword in the input header is 'SIMPLE', the 'XTENSION' keyword is replaced by the 'SIMPLE' keyword. This behavior is otherwise dumb as to whether or not the resulting header is a valid primary or extension header. This is mostly provided to support backwards compatibility with the old *Header.fromTxtFile* method, and only applies if `update=True`.

useblanks, bottom, end : bool, optional

These arguments are passed to *Header.append* while appending new cards to the header.

fromTxtFile(**args*, ***kwargs*)

Deprecated since version 0.1: This is equivalent to `self.extend(Header.fromtextfile(fileobj), update=True, update_first=True)`. Note that there there is no direct equivalent to the `replace=True` option since *Header.fromtextfile* returns a new *Header* instance.

Input the header parameters from an ASCII file.

The input header cards will be used to update the current header. Therefore, when an input card key matches a card key that already exists in the header, that card will be updated in place. Any input cards that do not already exist in the header will be added. Cards will not be deleted from the header.

Parameters

fileobj : file path, file object or file-like object

Input header parameters file.

replace : bool, optional

When *True*, indicates that the entire header should be replaced with the contents of the ASCII file instead of just updating the current header.

classmethod fromfile(*fileobj*, *sep=''*, *endcard=True*, *padding=True*)

Similar to *Header.fromstring*, but reads the header string from a given file-like object or filename.

Parameters

fileobj : str, file-like

A filename or an open file-like object from which a FITS header is to be read. For open file handles the file pointer must be at the beginning of the header.

sep : str, optional

The string separating cards from each other, such as a newline. By default there is no card separator (as is the case in a raw FITS file).

endcard : bool, optional

If True (the default) the header must end with an END card in order to be considered valid. If an END card is not found an *IOError* is raised.

padding : bool, optional

If True (the default) the header will be required to be padded out to a multiple of 2880, the FITS header block size. Otherwise any padding, or lack thereof, is ignored.

Returns

header

A new *Header* instance.

classmethod fromkeys (*iterable*, *value=None*)

Similar to dict.fromkeys–creates a new *Header* from an iterable of keywords and an optional default value.

This method is not likely to be particularly useful for creating real world FITS headers, but it is useful for testing.

Parameters

iterable

Any iterable that returns strings representing FITS keywords.

value : optional

A default value to assign to each keyword; must be a valid type for FITS keywords.

Returns

header

A new *Header* instance.

classmethod fromstring (*data*, *sep=''*)

Creates an HDU header from a byte string containing the entire header data.

Parameters

data : str

String containing the entire header.

sep : str, optional

The string separating cards from each other, such as a newline. By default there is no card separator (as is the case in a raw FITS file).

Returns

header

A new *Header* instance.

classmethod fromtextfile (*fileobj*, *endcard=False*)

Equivalent to:

```
>>> Header.fromfile(fileobj, sep='\n', endcard=False,
...                 padding=False)
```

get (*key*, *default=None*)

Similar to dict.get–returns the value associated with keyword in the header, or a default value if the keyword is not found.

Parameters

key : str

A keyword that may or may not be in the header.

default : optional

A default value to return if the keyword is not found in the header.

Returns

value

The value associated with the given keyword, or the default value if the keyword is not in the header.

get_comment (**args*, ***kwargs*)

Deprecated since version 0.1: The get_comment function is deprecated and may be removed in a future version. Use `header['COMMENT']` instead.

Get all comment cards as a list of string texts.

get_history (**args*, ***kwargs*)

Deprecated since version 0.1: The get_history function is deprecated and may be removed in a future version. Use `header['HISTORY']` instead.

Get all history cards as a list of string texts.

index (*keyword*, *start=None*, *stop=None*)

Returns the index if the first instance of the given keyword in the header, similar to *list.index* if the Header object is treated as a list of keywords.

Parameters

keyword : str

The keyword to look up in the list of all keywords in the header

start : int, optional

The lower bound for the index

stop : int, optional

The upper bound for the index

insert (*key*, *card*, *useblanks=True*, *after=False*)

Inserts a new keyword+value card into the Header at a given location, similar to *list.insert*.

Parameters

key : int, str, or tuple

The index into the list of header keywords before which the new keyword should be inserted, or the name of a keyword before which the new keyword should be inserted. Can also accept a (keyword, index) tuple for inserting around duplicate keywords.

card : str, tuple

A keyword or a (keyword, value, [comment]) tuple; see *Header.append*

useblanks : bool, optional

If there are blank cards at the end of the Header, replace the first blank card so that the total number of cards in the Header does not increase. Otherwise preserve the number of blank cards.

after : bool, optional

If set to *True*, insert *after* the specified index or keyword, rather than before it. Defaults to *False*.

items ()

Like `dict.items`.

iteritems()
Like dict.iteritems.

iterkeys()
Like dict.iterkeys–iterating directly over the *Header* instance has the same behavior.

itervalues()
Like dict.itervalues.

keys()
Return a list of keywords in the header in the order they appear–like dict.keys but ordered.

pop(**args*)
Works like list.pop if no arguments or an index argument are supplied; otherwise works like dict.pop.

popitem()
Similar to dict.popitem.

remove(*keyword*, *ignore_missing=False*, *remove_all=False*)
Removes the first instance of the given keyword from the header similar to *list.remove* if the Header object is treated as a list of keywords.

Parameters
keyword : str

The keyword of which to remove the first instance in the header.

ignore_missing : bool, optional

When True, ignores missing keywords. Otherwise, if the keyword is not present in the header a KeyError is raised.

remove_all : bool, optional

When True, all instances of keyword will be removed. Otherwise only the first instance of the given keyword is removed.

rename_key(**args*, ***kwargs*)
Deprecated since version 0.1: The rename_key function is deprecated and may be removed in a future version. Use *Header.rename_keyword* instead.

rename_keyword(*oldkeyword*, *newkeyword*, *force=False*)
Rename a card's keyword in the header.

Parameters
oldkeyword : str or int

Old keyword or card index

newkeyword : str

New keyword

force : bool, optional

When *True*, if the new keyword already exists in the header, force the creation of a duplicate keyword. Otherwise a *ValueError* is raised.

set(*keyword*, *value=None*, *comment=None*, *before=None*, *after=None*)
Set the value and/or comment and/or position of a specified keyword.

If the keyword does not already exist in the header, a new keyword is created in the specified position, or appended to the end of the header if no position is specified.

This method is similar to *Header.update* prior to PyFITS 3.1.

Note: It should be noted that `header.set(keyword, value)` and `header.set(keyword, value, comment)` are equivalent to `header[keyword] = value` and `header[keyword] = (value, comment)` respectively.

New keywords can also be inserted relative to existing keywords using, for example:

```
>>> header.insert('NAXIS1', ('NAXIS', 2, 'Number of axes'))
```

to insert before an existing keyword, or:

```
>>> header.insert('NAXIS', ('NAXIS1', 4096), after=True)
```

to insert after an existing keyword.

The only advantage of using *Header.set* is that it easily replaces the old usage of *Header.update* both conceptually and in terms of function signature.

Parameters

keyword : str

A header keyword

value : str, optional

The value to set for the given keyword; if None the existing value is kept, but '' may be used to set a blank value

comment : str, optional

The comment to set for the given keyword; if None the existing comment is kept, but `''` may be used to set a blank comment

before : str, int, optional

Name of the keyword, or index of the *Card* before which this card should be located in the header. The argument `before` takes precedence over `after` if both specified.

after : str, int, optional

Name of the keyword, or index of the *Card* after which this card should be located in the header.

setdefault(*key*, *default=None*)

Similar to `dict.setdefault`.

toTxtFile(**args*, ***kwargs*)

Deprecated since version 0.1: The toTxtFile function is deprecated and may be removed in a future version. Use *Header.totextfile* instead.

Output the header parameters to a file in ASCII format.

Parameters

fileobj : file path, file object or file-like object

Output header parameters file.

clobber : bool

When *True*, overwrite the output file if it exists.

tofile(*fileobj*, *sep=''*, *endcard=True*, *padding=True*, *clobber=False*)
Writes the header to file or file-like object.

By default this writes the header exactly as it would be written to a FITS file, with the END card included and padding to the next multiple of 2880 bytes. However, aspects of this may be controlled.

Parameters

fileobj : str, file, optional

Either the pathname of a file, or an open file handle or file-like object

sep : str, optional

The character or string with which to separate cards. By default there is no separator, but one could use `'\\n'`, for example, to separate each card with a new line

endcard : bool, optional

If *True* (default) adds the END card to the end of the header string

padding : bool, optional

If *True* (default) pads the string with spaces out to the next multiple of 2880 characters

clobber : bool, optional

If *True*, overwrites the output file if it already exists

tostring(*sep=''*, *endcard=True*, *padding=True*)
Returns a string representation of the header.

By default this uses no separator between cards, adds the END card, and pads the string with spaces to the next multiple of 2880 bytes. That is, it returns the header exactly as it would appear in a FITS file.

Parameters

sep : str, optional

The character or string with which to separate cards. By default there is no separator, but one could use `'\\n'`, for example, to separate each card with a new line

endcard : bool, optional

If True (default) adds the END card to the end of the header string

padding : bool, optional

If True (default) pads the string with spaces out to the next multiple of 2880 characters

Returns

s : str

A string representing a FITS header.

totextfile(*fileobj*, *endcard=False*, *clobber=False*)
Equivalent to:

```
>>> Header.tofile(fileobj, sep='\n', endcard=False,
...               padding=False, clobber=clobber)
```

update(**args*, ***kwargs*)
Update the Header with new keyword values, updating the values of existing keywords and appending new keywords otherwise; similar to *dict.update*.

update accepts either a dict-like object or an iterable. In the former case the keys must be header keywords and the values may be either scalar values or (value, comment) tuples. In the case of an iterable the items must be (keyword, value) tuples or (keyword, value, comment) tuples.

Arbitrary arguments are also accepted, in which case the update() is called again with the kwargs dict as its only argument. That is,

```
>>> header.update(NAXIS1=100, NAXIS2=100)
```

is equivalent to:

```
header.update({'NAXIS1': 100, 'NAXIS2': 100})
```

Warning: As this method works similarly to *dict.update* it is very different from the `Header.update()` method in PyFITS versions prior to 3.1.0. However, support for the old API is also maintained for backwards compatibility. If update() is called with at least two positional arguments then it can be assumed that the old API is being used. Use of the old API should be considered **deprecated**. Most uses of the old API can be replaced as follows:

•Replace

```
header.update(keyword, value)
```

with

```
header[keyword] = value
```

•Replace

```
header.update(keyword, value, comment=comment)
```

with

```
header[keyword] = (value, comment)
```

•Replace

```
header.update(keyword, value, before=before_keyword)
```

with

```
header.insert(before_keyword, (keyword, value))
```

•Replace

```
header.update(keyword, value, after=after_keyword)
```

with

```
header.insert(after_keyword, (keyword, value),
              after=True)
```

See also *`Header.set`* which is a new method that provides an interface similar to the old `Header.update()` and may help make transition a little easier.
For reference, the old documentation for the old `Header.update()` is provided below:

Update one header card.

If the keyword already exists, it's value and/or comment will be updated. If it does not exist, a new card will be created and it will be placed before or after the specified location. If no `before` or `after` is specified, it will be appended at the end.

Parameters

key : str

keyword

value : str

value to be used for updating

comment : str, optional

to be used for updating, default=None.

before : str, int, optional

name of the keyword, or index of the *Card* before which the new card will be placed. The argument `before` takes precedence over `after` if both specified.

after : str, int, optional

name of the keyword, or index of the *Card* after which the new card will be placed.

savecomment : bool, optional

When *True*, preserve the current comment for an existing keyword. The argument `savecomment` takes precedence over `comment` if both specified. If `comment` is not specified then the current comment will automatically be preserved.

values()

Returns a list of the values of all cards in the header.

ascard

Deprecated since version 0.1: The ascard function is deprecated and may be removed in a future version. Use the *cards* attribute instead.

Returns a `CardList` object wrapping this Header; provided for backwards compatibility for the old API (where Headers had an underlying `CardList`).

cards

The underlying physical cards that make up this Header; it can be looked at, but it should not be modified directly.

comments

View the comments associated with each keyword, if any.

For example, to see the comment on the NAXIS keyword:

```
>>> header.comments['NAXIS']
number of data axes
```

Comments can also be updated through this interface:

```
>>> header.comments['NAXIS'] = 'Number of data axes'
```

class `stsci.tools.stpyfits.`**ConstantValuePrimaryHDU**(*data=None*, *header=None*, *do_not_scale_image_data=False*, *uint=False*, ***kwargs*)

Bases: `stsci.tools.stpyfits._ConstantValueImageBaseHDU`, `astropy.io.fits.hdu.image.PrimaryHDU`

Primary HDUs with constant value arrays.

class `stsci.tools.stpyfits.`**ConstantValueImageHDU**(*data=None*, *header=None*, *do_not_scale_image_data=False*, *uint=False*, ***kwargs*)

Bases: `stsci.tools.stpyfits._ConstantValueImageBaseHDU`, `astropy.io.fits.hdu.image.ImageHDU`

Image extension HDUs with constant value arrays.

stsci.tools.stpyfits.**create_card**(**args*, ***kwargs*)
Deprecated since version 0.1: The create_card function is deprecated and may be removed in a future version. Use `Card.__init__` instead.

stsci.tools.stpyfits.**create_card_from_string**(**args*, ***kwargs*)
Deprecated since version 0.1: The create_card_from_string function is deprecated and may be removed in a future version. Use *Card.fromstring* instead.

Construct a *Card* object from a (raw) string. It will pad the string if it is not the length of a card image (80 columns). If the card image is longer than 80 columns, assume it contains `CONTINUE` card(s).

stsci.tools.stpyfits.**upper_key**(**args*, ***kwargs*)
Deprecated since version 0.1: The upper_key function is deprecated and may be removed in a future version. Use *Card.normalize_keyword* instead.

classmethod to convert a keyword value that may contain a field-specifier to uppercase. The effect is to raise the key to uppercase and leave the field specifier in its original case.

Parameters
key : or str

A keyword value or a `keyword.field-specifier` value

stsci.tools.stpyfits.**getheader**(**args*, ***kwargs*)
Get the header from an extension of a FITS file.

Parameters
filename : file path, file object, or file like object

File to get header from. If an opened file object, its mode must be one of the following rb, rb+, or ab+).

ext, extname, extver

The rest of the arguments are for extension specification. See the *getdata* documentation for explanations/examples.

kwargs

Any additional keyword arguments to be passed to `astropy.io.fits.open`.

Returns
header : *Header* object

stsci.tools.stpyfits.**getdata**(**args*, ***kwargs*)
Get the data from an extension of a FITS file (and optionally the header).

Parameters
filename : file path, file object, or file like object

File to get data from. If opened, mode must be one of the following rb, rb+, or ab+.

ext

The rest of the arguments are for extension specification. They are flexible and are best illustrated by examples.

No extra arguments implies the primary header:

```
getdata('in.fits')
```

By extension number:

```
getdata('in.fits', 0)      # the primary header
getdata('in.fits', 2)      # the second extension
getdata('in.fits', ext=2)  # the second extension
```

By name, i.e., EXTNAME value (if unique):

```
getdata('in.fits', 'sci')
getdata('in.fits', extname='sci')  # equivalent
```

Note EXTNAME values are not case sensitive

By combination of EXTNAME and EXTVER‘‘ as separate arguments or as a tuple:

```
getdata('in.fits', 'sci', 2)  # EXTNAME='SCI' & EXTVER=2
getdata('in.fits', extname='sci', extver=2)  # equivalent
getdata('in.fits', ('sci', 2))  # equivalent
```

Ambiguous or conflicting specifications will raise an exception:

```
getdata('in.fits', ext=('sci',1), extname='err', extver=2)
```

header : bool, optional

If *True*, return the data and the header of the specified HDU as a tuple.

lower, upper : bool, optional

If `lower` or `upper` are *True*, the field names in the returned data object will be converted to lower or upper case, respectively.

view : ndarray, optional

When given, the data will be returned wrapped in the given ndarray subclass by calling:

```
data.view(view)
```

kwargs

Any additional keyword arguments to be passed to `astropy.io.fits.open`.

Returns

array : array, record array or groups data object

Type depends on the type of the extension being referenced.

If the optional keyword `header` is set to *True*, this function will return a (`data`, `header`) tuple.

`stsci.tools.stpyfits.`**`getval`**(**args*, ***kwargs*)

Get a keyword's value from a header in a FITS file.

Parameters

filename : file path, file object, or file like object

Name of the FITS file, or file object (if opened, mode must be one of the following rb, rb+, or ab+).

keyword : str

Keyword name

ext, extname, extver

The rest of the arguments are for extension specification. See `getdata` for explanations/examples.

kwargs

Any additional keyword arguments to be passed to `astropy.io.fits.open`. *Note:* This function automatically specifies `do_not_scale_image_data = True` when opening the file so that values can be retrieved from the unmodified header.

Returns

keyword value : str, int, or float

`stsci.tools.stpyfits.`**`setval`**(**args*, ***kwargs*)

Set a keyword's value from a header in a FITS file.

If the keyword already exists, it's value/comment will be updated. If it does not exist, a new card will be created and it will be placed before or after the specified location. If no `before` or `after` is specified, it will be appended at the end.

When updating more than one keyword in a file, this convenience function is a much less efficient approach compared with opening the file for update, modifying the header, and closing the file.

Parameters

filename : file path, file object, or file like object

Name of the FITS file, or file object If opened, mode must be update (rb+). An opened file object or `GzipFile` object will be closed upon return.

keyword : str

Keyword name

value : str, int, float, optional

Keyword value (default: *None*, meaning don't modify)

comment : str, optional

Keyword comment, (default: *None*, meaning don't modify)

before : str, int, optional

Name of the keyword, or index of the card before which the new card will be placed. The argument `before` takes precedence over `after` if both are specified (default: *None*).

after : str, int, optional

Name of the keyword, or index of the card after which the new card will be placed. (default: *None*).

savecomment : bool, optional

When *True*, preserve the current comment for an existing keyword. The argument `savecomment` takes precedence over `comment` if both specified. If `comment` is not specified then the current comment will automatically be preserved (default: *False*).

ext, extname, extver

The rest of the arguments are for extension specification. See `getdata` for explanations/examples.

kwargs

Any additional keyword arguments to be passed to `astropy.io.fits.open`. *Note:* This function automatically specifies `do_not_scale_image_data = True` when opening the file so that values can be retrieved from the unmodified header.

`stsci.tools.stpyfits.`**`delval`**`(*args, **kwargs)`

Delete all instances of keyword from a header in a FITS file.

Parameters

filename : file path, file object, or file like object

Name of the FITS file, or file object If opened, mode must be update (rb+). An opened file object or `GzipFile` object will be closed upon return.

keyword : str, int

Keyword name or index

ext, extname, extver

The rest of the arguments are for extension specification. See *`getdata`* for explanations/examples.

kwargs

Any additional keyword arguments to be passed to `astropy.io.fits.open`. *Note:* This function automatically specifies `do_not_scale_image_data = True` when opening the file so that values can be retrieved from the unmodified header.

`stsci.tools.stpyfits.`**`writeto`**`(*args, **kwargs)`

Create a new FITS file using the supplied data/header.

Parameters

filename : file path, file object, or file like object

File to write to. If opened, must be opened in a writeable binary mode such as 'wb' or 'ab+'.

data : array, record array, or groups data object

data to write to the new file

header : *`Header`* object, optional

the header associated with `data`. If *None*, a header of the appropriate type is created for the supplied data. This argument is optional.

output_verify : str

Output verification option. Must be one of `"fix"`, `"silentfix"`, `"ignore"`, `"warn"`, or `"exception"`. May also be any combination of `"fix"` or `"silentfix"` with `"+ignore"`, `+warn`, or `+exception"` (e.g. ``"fix+warn"`). See verify for more info.

clobber : bool, optional

If *True*, and if filename already exists, it will overwrite the file. Default is *False*.

checksum : bool, optional

If *True*, adds both `DATASUM` and `CHECKSUM` cards to the headers of all HDU's written to the file.

`stsci.tools.stpyfits.`**`append`**`(*args, **kwargs)`

Append the header/data to FITS file if filename exists, create if not.

If only `data` is supplied, a minimal header is created.

Parameters

filename : file path, file object, or file like object

File to write to. If opened, must be opened for update (rb+) unless it is a new file, then it must be opened for append (ab+). A file or `GzipFile` object opened for update will be closed after return.

data : array, table, or group data object

the new data used for appending

header : *Header* object, optional

The header associated with `data`. If *None*, an appropriate header will be created for the data object supplied.

checksum : bool, optional

When *True* adds both `DATASUM` and `CHECKSUM` cards to the header of the HDU when written to the file.

verify : bool, optional

When *True*, the existing FITS file will be read in to verify it for correctness before appending. When *False*, content is simply appended to the end of the file. Setting `verify` to *False* can be much faster.

kwargs

Any additional keyword arguments to be passed to `astropy.io.fits.open`.

`stsci.tools.stpyfits.`**`update`**(**args*, ***kwargs*)

Update the specified extension with the input data/header.

Parameters

filename : file path, file object, or file like object

File to update. If opened, mode must be update (rb+). An opened file object or `GzipFile` object will be closed upon return.

data : array, table, or group data object

the new data used for updating

header : *Header* object, optional

The header associated with `data`. If *None*, an appropriate header will be created for the data object supplied.

ext, extname, extver

The rest of the arguments are flexible: the 3rd argument can be the header associated with the data. If the 3rd argument is not a *Header*, it (and other positional arguments) are assumed to be the extension specification(s). Header and extension specs can also be keyword arguments. For example:

```
update(file, dat, hdr, 'sci')  # update the 'sci' extension
update(file, dat, 3)  # update the 3rd extension
update(file, dat, hdr, 3)  # update the 3rd extension
update(file, dat, 'sci', 2)  # update the 2nd SCI extension
update(file, dat, 3, header=hdr)  # update the 3rd extension
update(file, dat, header=hdr, ext=5)  # update the 5th extension
```

kwargs

Any additional keyword arguments to be passed to `astropy.io.fits.open`.

`stsci.tools.stpyfits.`**`info`**(**args*, ***kwargs*)

Print the summary information on a FITS file.

This includes the name, type, length of header, data shape and type for each extension.

Parameters

filename : file path, file object, or file like object

FITS file to obtain info from. If opened, mode must be one of the following: rb, rb+, or ab+ (i.e. the file must be readable).

output : file, bool, optional

A file-like object to write the output to. If `False`, does not output to a file and instead returns a list of tuples representing the HDU info. Writes to `sys.stdout` by default.

kwargs

Any additional keyword arguments to be passed to `astropy.io.fits.open`. *Note:* This function sets `ignore_missing_end=True` by default.

`stsci.tools.stpyfits.`**`tdump`**(**args*, ***kwargs*)

Deprecated since version 0.1: The tdump function is deprecated and may be removed in a future version. Use *`tabledump`* instead.

`stsci.tools.stpyfits.`**`tcreate`**(**args*, ***kwargs*)

Deprecated since version 0.1: The tcreate function is deprecated and may be removed in a future version. Use *`tableload`* instead.

`stsci.tools.stpyfits.`**`tabledump`**(*filename*, *datafile=None*, *cdfile=None*, *hfile=None*, *ext=1*, *clobber=False*)

Dump a table HDU to a file in ASCII format. The table may be dumped in three separate files, one containing column definitions, one containing header parameters, and one for table data.

Parameters

filename : file path, file object or file-like object

Input fits file.

datafile : file path, file object or file-like object, optional

Output data file. The default is the root name of the input fits file appended with an underscore, followed by the extension number (ext), followed by the extension `.txt`.

cdfile : file path, file object or file-like object, optional

Output column definitions file. The default is *None*, no column definitions output is produced.

hfile : file path, file object or file-like object, optional

Output header parameters file. The default is *None*, no header parameters output is produced.

ext : int

The number of the extension containing the table HDU to be dumped.

clobber : bool

Overwrite the output files if they exist.

Notes

The primary use for the *tabledump* function is to allow editing in a standard text editor of the table data and parameters. The `tcreate` function can be used to reassemble the table from the three ASCII files.

- **datafile:** Each line of the data file represents one row of table data. The data is output one column at a time in column order. If a column contains an array, each element of the column array in the current row is output before moving on to the next column. Each row ends with a new line.

  Integer data is output right-justified in a 21-character field followed by a blank. Floating point data is output right justified using 'g' format in a 21-character field with 15 digits of precision, followed by a blank. String data that does not contain whitespace is output left-justified in a field whose width matches the width specified in the `TFORM` header parameter for the column, followed by a blank. When the string data contains whitespace characters, the string is enclosed in quotation marks (`""`). For the last data element in a row, the trailing blank in the field is replaced by a new line character.

  For column data containing variable length arrays ('P' format), the array data is preceded by the string `'VLA_Length= '` and the integer length of the array for that row, left-justified in a 21-character field, followed by a blank.

  Note: This format does *not* support variable length arrays using the ('Q' format) due to difficult to overcome ambiguities. What this means is that this file format cannot support VLA columns in tables stored in files that are over 2 GB in size.

  For column data representing a bit field ('X' format), each bit value in the field is output right-justified in a 21-character field as 1 (for true) or 0 (for false).

- **cdfile:** Each line of the column definitions file provides the definitions for one column in the table. The line is broken up into 8, sixteen-character fields. The first field provides the column name (`TTYPEn`). The second field provides the column format (`TFORMn`). The third field provides the display format (`TDISPn`). The fourth field provides the physical units (`TUNITn`). The fifth field provides the dimensions for a multidimensional array (`TDIMn`). The sixth field provides the value that signifies an undefined value (`TNULLn`). The seventh field provides the scale factor (`TSCALn`). The eighth field provides the offset value (`TZEROn`). A field value of `""` is used to represent the case where no value is provided.

- **hfile:** Each line of the header parameters file provides the definition of a single HDU header card as represented by the card image.

`stsci.tools.stpyfits.`**`tableload`**(*datafile*, *cdfile*, *hfile=None*)

Create a table from the input ASCII files. The input is from up to three separate files, one containing column definitions, one containing header parameters, and one containing column data. The header parameters file is not required. When the header parameters file is absent a minimal header is constructed.

Parameters

datafile : file path, file object or file-like object

Input data file containing the table data in ASCII format.

cdfile : file path, file object or file-like object

Input column definition file containing the names, formats, display formats, physical units, multidimensional array dimensions, undefined values, scale factors, and offsets associated with the columns in the table.

hfile : file path, file object or file-like object, optional

Input parameter definition file containing the header parameter definitions to be associated with the table. If *None*, a minimal header is constructed.

Notes

The primary use for the *tableload* function is to allow the input of ASCII data that was edited in a standard text editor of the table data and parameters. The tabledump function can be used to create the initial ASCII files.

- **datafile:** Each line of the data file represents one row of table data. The data is output one column at a time in column order. If a column contains an array, each element of the column array in the current row is output before moving on to the next column. Each row ends with a new line.

  Integer data is output right-justified in a 21-character field followed by a blank. Floating point data is output right justified using 'g' format in a 21-character field with 15 digits of precision, followed by a blank. String data that does not contain whitespace is output left-justified in a field whose width matches the width specified in the `TFORM` header parameter for the column, followed by a blank. When the string data contains whitespace characters, the string is enclosed in quotation marks (`""`). For the last data element in a row, the trailing blank in the field is replaced by a new line character.

  For column data containing variable length arrays ('P' format), the array data is preceded by the string `'VLA_Length= '` and the integer length of the array for that row, left-justified in a 21-character field, followed by a blank.

  Note: This format does *not* support variable length arrays using the ('Q' format) due to difficult to overcome ambiguities. What this means is that this file format cannot support VLA columns in tables stored in files that are over 2 GB in size.

  For column data representing a bit field ('X' format), each bit value in the field is output right-justified in a 21-character field as 1 (for true) or 0 (for false).

- **cdfile:** Each line of the column definitions file provides the definitions for one column in the table. The line is broken up into 8, sixteen-character fields. The first field provides the column name (`TTYPEn`). The second field provides the column format (`TFORMn`). The third field provides the display format (`TDISPn`). The fourth field provides the physical units (`TUNITn`). The fifth field provides the dimensions for a multidimensional array (`TDIMn`). The sixth field provides the value that signifies an undefined value (`TNULLn`). The seventh field provides the scale factor (`TSCALn`). The eighth field provides the offset value (`TZEROn`). A field value of `""` is used to represent the case where no value is provided.

- **hfile:** Each line of the header parameters file provides the definition of a single HDU header card as represented by the card image.

`stsci.tools.stpyfits.`**`table_to_hdu`**(*table*)

Convert an astropy.table.Table object to a FITS BinTableHDU

Parameters

table : astropy.table.Table

The table to convert.

Returns

table_hdu : astropy.io.fits.BinTableHDU

The FITS binary table HDU

`stsci.tools.stpyfits.`**`open`**(**args*, ***kwargs*)

Factory function to open a FITS file and return an *HDUList* object.

Parameters

name : file path, file object, file-like object or pathlib.Path object

File to be opened.

mode : str, optional

Open mode, 'readonly' (default), 'update', 'append', 'denywrite', or 'ostream'.

If `name` is a file object that is already opened, `mode` must match the mode the file was opened with, readonly (rb), update (rb+), append (ab+), ostream (w), denywrite (rb)).

memmap : bool, optional

Is memory mapping to be used?

save_backup : bool, optional

If the file was opened in update or append mode, this ensures that a backup of the original file is saved before any changes are flushed. The backup has the same name as the original file with ".bak" appended. If "file.bak" already exists then "file.bak.1" is used, and so on.

cache : bool, optional

If the file name is a URL, `download_file` is used to open the file. This specifies whether or not to save the file locally in Astropy's download cache (default: *True*).

kwargs : dict, optional

additional optional keyword arguments, possible values are:

- **uint** : bool

  Interpret signed integer data where `BZERO` is the central value and `BSCALE == 1` as unsigned integer data. For example, `int16` data with `BZERO = 32768` and `BSCALE = 1` would be treated as `uint16` data. This is enabled by default so that the pseudo-unsigned integer convention is assumed.

  Note, for backward compatibility, the kwarg **uint16** may be used instead. The kwarg was renamed when support was added for integers of any size.

- **ignore_missing_end** : bool

  Do not issue an exception when opening a file that is missing an `END` card in the last header.

- **checksum** : bool, str

  If *True*, verifies that both `DATASUM` and `CHECKSUM` card values (when present in the HDU header) match the header and data of all HDU's in the file. Updates to a file that already has a checksum will preserve and update the existing checksums unless this argument is given a value of 'remove', in which case the CHECKSUM and DATASUM values are not checked, and are removed when saving changes to the file.

- **disable_image_compression** : bool

  If *True*, treats compressed image HDU's like normal binary table HDU's.

- **do_not_scale_image_data** : bool

  If *True*, image data is not scaled using BSCALE/BZERO values when read.

- **ignore_blank** : bool

  If *True*, the BLANK keyword is ignored if present.

- **scale_back** : bool

  If *True*, when saving changes to a file that contained scaled image data, restore the data to the original type and reapply the original BSCALE/BZERO values.

This could lead to loss of accuracy if scaling back to integer values after performing floating point operations on the data.

Returns

hdulist : an *HDUList* object

HDUList containing all of the header data units in the file.

stsci.tools.stpyfits.**new_table**(**args*, ***kwargs*)

Deprecated since version 0.4: The new_table function is deprecated and may be removed in a future version. Use `BinTableHDU.from_columns` for new BINARY tables or `TableHDU.from_columns` for new ASCII tables instead.

Create a new table from the input column definitions.

Warning: Creating a new table using this method creates an in-memory *copy* of all the column arrays in the input. This is because if they are separate arrays they must be combined into a single contiguous array.

If the column data is already in a single contiguous array (such as an existing record array) it may be better to create a *BinTableHDU* instance directly. See the Astropy documentation for more details.

Parameters

input : sequence of *Column* or a *ColDefs*

The data to create a table from

header : *Header* instance

Header to be used to populate the non-required keywords

nrows : int

Number of rows in the new table

fill : bool

If *True*, will fill all cells with zeros or blanks. If *False*, copy the data from input, undefined cells will still be filled with zeros/blanks.

tbtype : str or type

Table type to be created (*BinTableHDU* or *TableHDU*) or the class name as a string. Currently only *BinTableHDU* and *TableHDU* (ASCII tables) are supported.

stsci.tools.stpyfits.**enable_stpyfits**()

stsci.tools.stpyfits.**disable_stpyfits**()

stsci.tools.stpyfits.**with_stpyfits**(*func*)

2.2 FITSDIFF

This module serves as a large library of helpful file operations, both for I/O of files and to extract information about the files. fitsdiff is now a part of PyFITS–the fitsdiff in PyFITS replaces the fitsdiff that used to be in the module.

Now this module just provides a wrapper around astropy.io.fits.diff for backwards compatibility with the old interface in case anyone uses it.

stsci.tools.fitsdiff.**list_parse**(*name_list*)

Parse a comma-separated list of values, or a filename (starting with @) containing a list value on each line.

2.3 WCSUTIL

The *wcsutil* module provides a stand-alone implementation of a WCS object which provides a number of basic transformations and query methods. Most (if not all) of these functions can be obtained from the use of the PyWCS or STWCS WCS object if those packages have been installed.

class stsci.tools.wcsutil.**WCSObject**(*rootname*, *header=None*, *shape=None*, *pa_key='PA_V3'*, *new=False*, *prefix=None*)

This class should contain the WCS information from the input exposure's header and provide conversion functionality from pixels to RA/Dec and back.

Syntax

The basic syntax for using this object is:

```
>>> wcs = wcsutil.WCSObject(rootname,header=None,shape=None,
>>>                         pa_key='PA_V3',new=no,prefix=None)
```

This will create a WCSObject which provides basic WCS functions.

Parameters

rootname: string

filename in a format supported by IRAF, specifically:

```
filename.hhh[group] -or-
filename.fits[ext] -or-
filename.fits[extname,extver]
```

header: object

PyFITS header object from which WCS keywords can be read

shape: tuple

tuple giving (nx,ny,pscale)

pa_key: string

name of keyword to read in telescopy orientation

new: boolean

specify a new object rather than creating one by reading in keywords from an existing image

prefix: string

string to use as prefix for creating archived versions of WCS keywords, if such keywords do not already exist

Notes

Setting 'new=yes' will create a WCSObject from scratch regardless of any input rootname. This avoids unexpected filename collisions.

Methods

print_archive(format=True)	print out archive keyword values
get_archivekw(keyword)	return archived value for WCS keyword
set_pscale()	set pscale attribute for object
compute_pscale(cd11,cd21)	compute pscale value
get_orient()	return orient computed from CD matrix
updateWCS(pixel_scale=None,orient=None,refpos=None,refval=None,size=None)	reset entire WCS based on given values
xy2rd(pos)	compute RA/Dec position for given (x,y) tuple
rd2xy(skypos,hour=no)	compute X,Y position for given (RA,Dec)
rotateCD(orient)	rotate CD matrix to new orientation given by 'orient'
recenter()	Reset reference position to X,Y center of frame
write(fitsname=None,archive=True,overwrite=False,quiet=True)	write out values of WCS to specified file
restore()	reset WCS keyword values to those from archived values
read_archive(header,prepend=None)	read any archive WCS keywords from PyFITS header
archive(prepend=None,overwrite=no,quiet=yes)	create archived copies of WCS keywords.
write_archive(fitsname=None,overwrite=no,quiet=yes)	write out the archived WCS values to the file
restoreWCS(prepend=None)	resets WCS values in file to original values
createReferenceWCS(refname,overwrite=yes)	write out values of WCS keywords to NEW FITS file without any image data
copy(deep=True)	create a copy of the WCSObject.
help()	prints out this help message

archive(*prepend=None*, *overwrite=False*, *quiet=True*)
Create backup copies of the WCS keywords with the given prepended string. If backup keywords are already present, only update them if 'overwrite' is set to 'yes', otherwise, do warn the user and do nothing. Set the WCSDATE at this time as well.

compute_pscale(*cd11*, *cd21*)
Compute the pixel scale based on active WCS values.

copy(*deep=True*)
Makes a (deep)copy of this object for use by other objects.

createReferenceWCS(*refname*, *overwrite=True*)
Write out the values of the WCS keywords to the NEW specified image 'fitsname'.

createWcsHDU()
Generate a WCS header object that can be used to populate a reference WCS HDU.

get_archivekw(*keyword*)
Return an archived/backup value for the keyword.

get_orient()
Return the computed orientation based on CD matrix.

help()
Prints out help message.

print_archive(*format=True*)
Prints out archived WCS keywords.

rd2xy(*skypos*, *hour=False*)
This method would use the WCS keywords to compute the XY position from a given RA/Dec tuple (in deg).

NOTE: Investigate how to let this function accept arrays as well as single positions. WJH 27Mar03

read_archive(*header*, *prepend=None*)
Extract a copy of WCS keywords from an open file header, if they have already been created and remember the prefix used for those keywords. Otherwise, setup the current WCS keywords as the archive values.

recenter()
Reset the reference position values to correspond to the center of the reference frame. Algorithm used here developed by Colin Cox - 27-Jan-2004.

restore()
Reset the active WCS keywords to values stored in the backup keywords.

restoreWCS(*prepend=None*)
Resets the WCS values to the original values stored in the backup keywords recorded in self.backup.

rotateCD(*orient*)
Rotates WCS CD matrix to new orientation given by 'orient'

scale_WCS(*pixel_scale*, *retain=True*)
Scale the WCS to a new pixel_scale. The 'retain' parameter [default value: True] controls whether or not to retain the original distortion solution in the CD matrix.

set_orient()
Return the computed orientation based on CD matrix.

set_pscale()
Compute the pixel scale based on active WCS values.

update()
Update computed values of WCS based on current CD matrix.

updateWCS(*pixel_scale=None*, *orient=None*, *refpos=None*, *refval=None*, *size=None*)
Create a new CD Matrix from the absolute pixel scale and reference image orientation.

write(*fitsname=None*, *wcs=None*, *archive=True*, *overwrite=False*, *quiet=True*)
Write out the values of the WCS keywords to the specified image.

If it is a GEIS image and 'fitsname' has been provided, it will automatically make a multi-extension FITS copy of the GEIS and update that file. Otherwise, it throw an Exception if the user attempts to directly update a GEIS image header.

If archive=True, also write out archived WCS keyword values to file. If overwrite=True, replace archived WCS values in file with new values.

If a WCSObject is passed through the 'wcs' keyword, then the WCS keywords of this object are copied to the header of the image to be updated. A use case fo rthis is updating the WCS of a WFPC2 data quality (_c1h.fits) file in order to be in sync with the science (_c0h.fits) file.

write_archive(*fitsname=None*, *overwrite=False*, *quiet=True*)
Saves a copy of the WCS keywords from the image header as new keywords with the user-supplied 'prepend' character(s) prepended to the old keyword names.

If the file is a GEIS image and 'fitsname' != None, create a FITS copy and update that version; otherwise, raise an Exception and do not update anything.

xy2rd(*pos*)
This method would apply the WCS keywords to a position to generate a new sky position.

The algorithm comes directly from 'imgtools.xy2rd'

translate (x,y) to (ra, dec)

stsci.tools.wcsutil.**ddtohms**(*xsky*, *ysky*, *verbose=False*)
Convert sky position(s) from decimal degrees to HMS format.

stsci.tools.wcsutil.**help**()

stsci.tools.wcsutil.**troll**(*roll*, *dec*, *v2*, *v3*)
Computes the roll angle at the target position based on:

```
    the roll angle at the V1 axis(roll),
    the dec of the target(dec), and
    the V2/V3 position of the aperture (v2,v3) in arcseconds.
```

Based on the algorithm provided by Colin Cox that is used in Generic Conversion at STScI.

2.4 Conversion Utilities

2.4.1 Convertwaiveredfits

__version__ = "1.0 (31 January, 2008)"

stsci.tools.convertwaiveredfits.**convertwaiveredfits**(*waiveredObject*, *outputFileName=None*, *forceFileOutput=False*, *convertTo='multiExtension'*, *verbose=False*)
Convert the input waivered FITS object to various formats. The default conversion format is multi-extension FITS. Generate an output file in the desired format if requested.

Parameters:

waiveredObject input object representing a waivered FITS file;
either a astropy.io.fits.HDUList object, a file object, or a file specification

outputFileName file specification for the output file
Default: None - do not generate an output file

forceFileOutput force the generation of an output file when the
outputFileName parameter is None; the output file specification will be the same as the input file specification with the last character of the base name replaced with the character *h* in multi-extension FITS format.

Default: False

convertTo target conversion type
Default: 'multiExtension'

verbose provide verbose output
Default: False

Returns:

hdul an HDUList object in the requested format.

Exceptions:

ValueError Conversion type is unknown

stsci.tools.convertwaiveredfits.**main**()

stsci.tools.convertwaiveredfits.**toMultiExtensionFits**(*waiveredObject*, *multiExtensionFileName=None*, *forceFileOutput=False*, *verbose=False*)

Convert the input waivered FITS object to a multi-extension FITS HDUList object. Generate an output multi-extension FITS file if requested.

Parameters:

waiveredObject input object representing a waivered FITS file;
either a astroyp.io.fits.HDUList object, a file object, or a file specification

outputFileName file specification for the output file
Default: None - do not generate an output file

forceFileOutput force the generation of an output file when the
outputFileName parameter is None; the output file specification will be the same as the input file specification with the last character of the base name replaced with the character 'h'. Default: False

verbose provide verbose output
Default: False

Returns:

mhdul an HDUList object in multi-extension FITS format.

Exceptions:

TypeError Input object is not a HDUList, a file object or a
file name

2.4.2 ReadGEIS

readgeis: Read GEIS file and convert it to a FITS extension file.

License: http://www.stsci.edu/resources/software_hardware/pyraf/LICENSE

Usage:

readgeis.py [options] GEISname FITSname

GEISname is the input GEIS file in GEIS format, and FITSname is the output file in FITS format. GEISname can be a directory name. In this case, it will try to use all **.??h* files as input file names.

If FITSname is omitted or is a directory name, this task will try to construct the output names from the input names, i.e.:

abc.xyh will have an output name of abc_xyf.fits

Options

-h print the help (this text)

Example

If used in Pythons script, a user can, e. g.:

```
>>> import readgeis
>>> hdulist = readgeis.readgeis(GEISFileName)
```

```
(do whatever with hdulist)
>>> hdulist.writeto(FITSFileName)
```

The most basic usage from the command line:

```
readgeis.py test1.hhh test1.fits
```

This command will convert the input GEIS file test1.hhh to a FITS file test1.fits.

From the command line:

```
readgeis.py .
```

this will convert all **.??h* files in the current directory to FITS files (of corresponding names) and write them in the current directory.

Another example of usage from the command line:

```
readgeis.py "u*" "*"
```

this will convert all *u*.??h* files in the current directory to FITS files (of corresponding names) and write them in the current directory. Note that when using wild cards, it is necessary to put them in quotes.

stsci.tools.readgeis.**parse_path**(*f1*, *f2*)
Parse two input arguments and return two lists of file names

stsci.tools.readgeis.**readgeis**(*input*)
Input GEIS files "input" will be read and a HDUList object will be returned.

The user can use the writeto method to write the HDUList object to a FITS file.

stsci.tools.readgeis.**stsci**(*hdulist*)
For STScI GEIS files, need to do extra steps.

stsci.tools.readgeis.**stsci2**(*hdulist*, *filename*)
For STScI GEIS files, need to do extra steps.

2.4.3 Check_files

The *check_files* module provides functions to perform verification of input file formats for use in betadrizzle. This set of functions also includes format conversion functions to convert GEIS or waiver-FITS HST images into multi-extension FITS (MEF) files.

stsci.tools.check_files.**checkFITSFormat**(*filelist*, *ivmlist=None*)
This code will check whether or not files are GEIS or WAIVER FITS and convert them to MEF if found. It also keeps the IVMLIST consistent with the input filelist, in the case that some inputs get dropped during the check/conversion.

stsci.tools.check_files.**checkFiles**(*filelist*, *ivmlist=None*)

- Converts waiver fits sciece and data quality files to MEF format
- Converts GEIS science and data quality files to MEF format
- Checks for stis association tables and splits them into single imsets
- Removes files with EXPTIME=0 and the corresponding ivm files
- Removes files with NGOODPIX == 0 (to exclude saturated images)
- Removes files with missing PA_V3 keyword

The list of science files should match the list of ivm files at the end.

stsci.tools.check_files.**checkNGOODPIX**(*filelist*)
Only for ACS, WFC3 and STIS, check NGOODPIX If all pixels are 'bad' on all chips, exclude this image from further processing. Similar checks requiring comparing 'driz_sep_bits' against WFPC2 c1f.fits arrays and NICMOS DQ arrays will need to be done separately (and later).

stsci.tools.check_files.**checkPA_V3**(*fnames*)

stsci.tools.check_files.**checkStisFiles**(*filelist*, *ivmlist=None*)

stsci.tools.check_files.**check_exptime**(*filelist*)
Removes files with EXPTIME==0 from filelist.

stsci.tools.check_files.**convert2fits**(*sci_ivm*)
Checks if a file is in WAIVER of GEIS format and converts it to MEF

stsci.tools.check_files.**countInput**(*input*)

stsci.tools.check_files.**geis2mef**(*sciname*, *convert_dq=True*)
Converts a GEIS science file and its corresponding data quality file (if present) to MEF format Writes out both files to disk. Returns the new name of the science image.

stsci.tools.check_files.**isSTISSpectroscopic**(*fname*)

stsci.tools.check_files.**splitStis**(*stisfile*, *sci_count*)

Purpose
Split a STIS association file into multiple imset MEF files.

Split the corresponding spt file if present into single spt files. If an spt file can't be split or is missing a Warning is printed.

Returns
names: list

a list with the names of the new flt files.

stsci.tools.check_files.**stisExt2PrimKw**(*stisfiles*)
Several kw which are usuall yin the primary header are in the extension header for STIS. They are copied to the primary header for convenience. List if kw: 'DATE-OBS', 'EXPEND', 'EXPSTART', 'EXPTIME'

stsci.tools.check_files.**stisObsCount**(*input*)
Input: A stis multiextension file Output: Number of stis science extensions in input

stsci.tools.check_files.**update_input**(*filelist*, *ivmlist=None*, *removed_files=None*)
Removes files flagged to be removed from the input filelist. Removes the corresponding ivm files if present.

stsci.tools.check_files.**waiver2mef**(*sciname*, *newname=None*, *convert_dq=True*)
Converts a GEIS science file and its corresponding data quality file (if present) to MEF format Writes out both files to disk. Returns the new name of the science image.

2.5 Association File Interpretation

Association files serve as FITS tables which relate a set of input files to the generation of calibrated or combined products. A module which provides utilities for reading, writing, creating and updating association tables and shift

files.

author
Warren Hack, Nadia Dencheva

version
'0.1 (2008-01-03)'

class stsci.tools.asnutil.**ASNMember**(*xoff=0.0*, *yoff=0.0*, *rot=0.0*, *xshift=0.0*, *yshift=0.0*, *scale=1.0*, *dshift=False*, *abshift=False*, *refimage=''*, *shift_frame=''*, *shift_units='pixels'*, *row=0*)

A dictionary like object representing a member of an association table. It looks like this:

```
'j8bt06nzq': {'abshift': False,
              'dshift': True,
              'refimage': 'j8bt06010_shifts_asn.fits[wcs]',
              'rot': 359.99829,
              'row': 1,
              'scale': 1.000165,
              'shift_frame': 'input',
              'shift_units': 'pixels',
              'xoff': 0.0,
              'xshift': 0.4091132,
              'yoff': 0.0,
              'yshift': -0.56702018}
```

If *abshift* is True, shifts, rotation and scale refer to absolute shifts. If *dshift* is True, they are delta shifts.

class stsci.tools.asnutil.**ASNTable**(*inlist=None*, *output=None*, *shiftfile=None*)

A dictionary like object which represents an association table. An ASNTable object looks like this:

```
{'members':
        {'j8bt06nyq': {'abshift': False,
                       'dshift': True,
                       'refimage': 'j8bt06010_shifts_asn.fits[wcs]',
                       'rot': 0.0,
                       'row': 0,
                       'scale': 1.0,
                       'shift_frame': 'input',
                       'shift_units': 'pixels',
                       'xoff': 0.0,
                       'xshift': 0.0,
                       'yoff': 0.0,
                       'yshift': 0.0},
         'j8bt06nzq': {'abshift': False,
                       'dshift': True,
                       'refimage': 'j8bt06010_shifts_asn.fits[wcs]',
                       'rot': 359.99829,
                       'row': 1,
                       'scale': 1.000165,
                       'shift_frame': 'input',
                       'shift_units': 'pixels',
                       'xoff': 0.0,
                       'xshift': 0.4091132,
                       'yoff': 0.0,
                       'yshift': -0.56702018}},
         'order': ['j8bt06nyq', 'j8bt06nzq'],
         'output': 'j8bt06nyq'}
```

Examples

Creating an ASNTable object from 3 filenames and a shift file would be done using:

```
>>> asnt=ASNTable([fname1,fname2,  fname3], shiftfile='shifts.txt')
```

The ASNTable object would have the 'members' and 'order' in the association table populated based on *infiles* and *shiftfile*.

This creates a blank association table from the ASNTable object:

```
>>> asnt.create()
```

Parameters

inlist : list

A list of filenames.

output : str

A user specified output name or 'final'.

shiftfile : str

A name of a shift file, if given, the association table will be updated with the values in the shift file.

buildPrimary(*fasn*, *output=None*)

create(*shiftfile=None*)

update(*members=None*, *shiftfile=None*, *replace=False*)

write(*output=None*)

Write association table to a file.

class stsci.tools.asnutil.**ShiftFile**(*filename=''*, *form='delta'*, *frame=None*, *units='pixels'*, *order=None*, *refimage=None*, ***kw*)

A shift file has the following format (name, Xsh, Ysh, Rot, Scale):

```
# frame: output
# refimage: tweak_wcs.fits[wcs]
# form: delta
# units: pixels
j8bt06nyq_flt.fits    0.0  0.0    0.0    1.0
j8bt06nzq_flt.fits    0.4091132  -0.5670202    359.9983    1.000165
```

This object creates a *dict* like object representing a shift file used by Pydrizzle and Mirashift.

Purpose

Create a dict like ShiftFile object from a shift file on disk or from variables in memory. If a file name is provided all other parameters are ignored.

Parameters

filename : str

Name of shift file on disk, see above the expected format

form : str

Form of shifts (absolute|delta)

frame : str

Frame in which the shifts should be applied (input|output)

units : str

Units in which the shifts are measured.

order : list

Keeps track of the order of the files.

refimage : str

name of reference image

****d**
[dict] keys: file names values: a list: [Xsh, Ysh, Rot, Scale] The keys must match the files in the order parameter.

Raises
ValueError

If reference file can't be found

Examples

These examples demonstrate a couple of the most common usages.

Read a shift file on disk using:

```
>>> sdict = ShiftFile('shifts.txt')
```

Pass values for the fields of the shift file and a dictionary with all files:

```
>>> d={'j8bt06nyq_flt.fits': [0.0, 0.0, 0.0, 1.0],
```

'j8bt06nzq_flt.fits': [0.4091132, -0.5670202, 359.9983, 1.000165]}

```
>>> sdict = ShiftFile(form='absolute', frame='output', units='pixels', order=['j8bt06nyq_flt.fit
                      'j8bt06nzq_flt.fits'], refimage='tweak_wcs.fits[wcs]', **d)
```

The return value can then be used to provide the shift information to code in memory.

readShiftFile (*filename*)
Reads a shift file from disk and populates a dictionary.

verifyShiftFile ()
Verifies that reference file exists.

writeShiftFile (*filename='shifts.txt'*)
Writes a shift file object to a file on disk using the convention for shift file format.

stsci.tools.asnutil.**readASNTable** (*fname*, *output=None*, *prodonly=False*)
Given a fits filename repesenting an association table reads in the table as a dictionary which can be used by pydrizzle and multidrizzle.

An association table is a FITS binary table with 2 required columns: 'MEMNAME', 'MEMTYPE'. It checks 'MEMPRSNT' column and removes all files for which its value is 'no'.

Parameters
fname : str

name of association table

output : str

name of output product - if not specified by the user, the first PROD-DTH name is used if present, if not, the first PROD-RPT name is used if present, if not, the rootname of the input association table is used.

prodonly : bool

what files should be considered as input if True - select only MEMTYPE=PROD* as input if False - select only MEMTYPE=EXP as input

Returns

asndict : dict

A dictionary-like object with all the association information.

Examples

An association table can be read from a file using the following commands:

```
>>> from stsci.tools import asnutil
>>> asntab = asnutil.readASNTable('j8bt06010_shifts_asn.fits', prodonly=False)
```

The *asntab* object can now be passed to other code to provide relationships between input and output images defined by the association table.

CHAPTER

THREE

IMAGE ACCESS MODULES

These modules all provide the capability to access sections of a FITS image using a scrolling buffer. License: http://www.stsci.edu/resources/software_hardware/pyraf/LICENSE License: http://www.stsci.edu/resources/software_hardware/pyraf/LICENSE

stsci.tools.nimageiter.**FileIter**(*filelist*, *bufsize=1024000*, *overlap=0*)

Return image section for each image listed on input, with the object performing the file I/O upon each call to the iterator.

The inputs can either be a single image or a list of them, with the return value matching the input type. All images in a list MUST have the same shape, though, in order for the iterator to scroll through them properly.

The size of section gets defined by 'bufsize'. The 'overlap' parameter provides a way of scrolling through the image with this many rows of overlap, with the default being no overlap at all.

stsci.tools.nimageiter.**ImageIter**(*imglist*, *bufsize=1024000*, *overlap=0*, *copy=0*, *updateSection=None*)

Return image section for each image listed on input. The inputs can either be a single image or a list of them, with the return value matching the input type. All images in a list MUST have the same shape, though, in order for the iterator to scroll through them properly.

The size of section gets defined by 'bufsize', while 'copy' specifies whether to return an explicit copy of each input section or simply return views. The 'overlap' parameter provides a way of scrolling through the image with this many rows of overlap, with the default being no overlap at all.

stsci.tools.nimageiter.**computeBuffRows**(*imgarr*, *bufsize=1024000*)

Function to compute the number of rows from the input array that fits in the allocated memory given by the bufsize.

stsci.tools.nimageiter.**computeNumberBuff**(*numrows*, *buffrows*, *overlap*)

Function to compute the number of buffer sections that will be used to read the input image given the specified overlap.

License: http://www.stsci.edu/resources/software_hardware/pyraf/LICENSE

class stsci.tools.iterfile.**IterFitsFile**(*name*)

This class defines an object which can be used to access the data from a FITS file without leaving the file-handle open between reads.

close()

Closes file handle for this FITS object.

open()

Opens the file for subsequent access.

set_inmemory(*val*)

Sets inmemory attribute to either True or False

type()
Returns the shape of the data array associated with this file.

stsci.tools.iterfile.**parseFilename**(*filename*)
Parse out filename from any specified extensions. Returns rootname and string version of extension name.

Modified from 'pydrizzle.fileutil' to allow this module to be independent of PyDrizzle/MultiDrizzle.

CHAPTER

FOUR

DATA ANALYSIS ROUTINES

These modules provide basic data analysis or data fitting functionality.

4.1 linefit

Fit a line to a data set with optional weights.

Returns the parameters of the model, bo, b1: Y = b0 + b1* X

author
Nadia Dencheva

version
'1.0 (2007-02-20)'

`stsci.tools.linefit.`**`linefit`**(*x*, *y*, *weights=None*)

Parameters

y: 1D numpy array

The data to be fitted

x: 1D numpy array

The x values of the y array. x and y must have the same shape.

weights: 1D numpy array, must have the same shape as x and y

weight values

Examples

```
>>> x=N.array([-5, -4 ,-3 ,-2 ,-1, 0, 1, 2, 3, 4, 5])
>>> y=N.array([1, 5, 4, 7, 10, 8, 9, 13, 14, 13, 18])
>>> around(linefit(x,y), decimals=5)
array([ 9.27273,  1.43636])
>>> x=N.array([1.3,1.3,2.0,2.0,2.7,3.3,3.3,3.7,3.7,4.,4.,4.,4.7,4.7,5.,5.3,5.3,5.3,5.7,6.,6.,6.3
>>> y = N.array([2.3,1.8,2.8,1.5,2.2,3.8,1.8,3.7,1.7,2.8,2.8,2.2,3.2,1.9,1.8,3.5,2.8,2.1,3.4,3.2
>>> around(linefit(x,y), decimals=5)
array([ 1.42564,  0.31579])
```

4.2 nmpfit

Python/Numeric version of this module was called mpfit. This version was modified to use numpy.

4.3 xyinterp

Module
xyinterp.py

Interpolates y based on the given xval.

x and *y* are a pair of independent/dependent variable arrays that must be the same length. The x array must also be sorted. *xval* is a user-specified value. This routine looks up *xval* in the x array and uses that information to properly interpolate the value in the y array.

author
Vicki Laidler

version
'0.1 (2006-07-06)'

`stsci.tools.xyinterp.`**`xyinterp`**(*x*, *y*, *xval*)

Purpose
Interpolates y based on the given xval.

x and y are a pair of independent/dependent variable arrays that must be the same length. The x array must also be sorted. xval is a user-specified value. This routine looks up xval in the x array and uses that information to properly interpolate the value in the y array.

Parameters

x: 1D numpy array

independent variable array: MUST BE SORTED

y: 1D numpy array

dependent variable array

xval: float

the x value at which you want to know the value of y

Returns

y: float

the value of y corresponding to xval

Raises

ValueError:

If arrays are unequal length; or x array is unsorted; or if xval falls outside the bounds of x (extrapolation is unsupported

:version: 0.1 last modified 2006-07-06

See also:

`numpy`

Notes

Use the searchsorted method on the X array to determine the bin in which xval falls; then use that information to compute the corresponding y value.

4.4 gfit

Return the gaussian fit of a 1D array.

Uses mpfit.py - a python implementation of the Levenberg-Marquardt least-squares minimization, based on MINPACK-1. See nmpfit.py for the history of this module (fortran -> idl -> python). nmpfit.py is a version of mpfit.py which uses numarray.

@author: Nadia Dencheva @version: '1.0 (2007-02-20)'

stsci.tools.gfit.**gfit1d**(*y*, *x=None*, *err=None*, *weights=None*, *par=None*, *parinfo=None*, *maxiter=200*, *quiet=0*)

Return the gaussian fit as an object.

Parameters

y: 1D Numarray array

The data to be fitted

x: 1D Numarray array

(optional) The x values of the y array. x and y must have the same shape.

err: 1D Numarray array

(optional) 1D array with measurement errors, must be the same shape as y

weights: 1D Numarray array

(optiional) 1D array with weights, must be the same shape as y

par: List

(optional) Starting values for the parameters to be fitted

parinfo: Dictionary of lists

(optional) provides additional information for the parameters. For a detailed description see nmpfit.py. Parinfo can be used to limit parameters or keep some of them fixed.

maxiter: number

Maximum number of iterations to perform Default: 200

quiet: number

if set to 1, nmpfit does not print to the screen Default: 0

Examples

```
>>> x=N.arange(10,20, 0.1)
>>> y= 10*N.e**(-(x-15)**2/4)
>>> print gfit1d(y,x=x, maxiter=20,quiet=1).params
[ 10.          15.           1.41421356]
```

4.4.1 Image Combination Modules

The *numcombine* module serves as a limited replacement for IRAF's 'imcombine' task.

class stsci.image.numcombine.**numCombine**(*arrObjectList*, *numarrayMaskList=None*, *combinationType='median'*, *nlow=0*, *nhigh=0*, *nkeep=1*, *upper=None*, *lower=None*)

A lite version of the imcombine IRAF task

Parameters

arrObjectList : list of ndarray

A sequence of inputs arrays, which are nominally a stack of identically shaped images.

numarrayMaskList : list of ndarray

A sequence of mask arrays to use for masking out 'bad' pixels from the combination The ndarray should be a numpy array, despite the variable name.

combinationType : {'median','imedian','iaverage','mean','sum','minimum'}

Type of operation should be used to combine the images The 'imedian' and 'iaverage' types ignore pixels which have been flagged as bad in all input arrays and returns the value from the last image in the stack for that pixel.

nlow : int [Default: 0]

Number of low pixels to throw out of the median calculation

nhigh : int [Default: 0]

Number of high pixels to throw out of the median calculation

nkeep : int [Default: 1]

Minimun number of pixels to keep for a valid computation

upper : float, optional

Throw out values >= to upper in a median calculation

lower: float, optional

Throw out values < lower in a median calculation

Returns

combArrObj : ndarray

The attribute '.combArrObj' holds the combined output array.

Examples

This class can be used to create a median image from a stack of images with the following commands:

```
>>> import numpy as np
>>> from stsci.image import numcombine as nc
>>> a = np.ones([5,5],np.float32)
>>> b = a - 0.05
>>> c = a + 0.1
>>> result = nc.numCombine([a,b,c],combinationType='mean')
>>> result.combArrObj
array([[ 1.01666665,  1.01666665,  1.01666665,  1.01666665,  1.01666665],
       [ 1.01666665,  1.01666665,  1.01666665,  1.01666665,  1.01666665],
       [ 1.01666665,  1.01666665,  1.01666665,  1.01666665,  1.01666665],
       [ 1.01666665,  1.01666665,  1.01666665,  1.01666665,  1.01666665],
       [ 1.01666665,  1.01666665,  1.01666665,  1.01666665,  1.01666665]], dtype=float32)
```

CHAPTER

FIVE

UTILITY FUNCTIONS FOR HANDLING BIT MASKS AND MASK ARRAYS.

A module that provides functions for manipulating bitmasks and data quality (DQ) arrays.

Authors
Mihai Cara (contact: help@stsci.edu)

License
http://www.stsci.edu/resources/software_hardware/pyraf/LICENSE

stsci.tools.bitmask.**bitmask2mask**(*bitmask*, *ignore_bits*, *good_pix_value=1*, *dtype=numpy.uint8*)

Interprets an array of bit flags and converts it to a "binary" mask array. This function is particularly useful to convert data quality arrays to binary masks.

Parameters

bitmask : numpy.ndarray

An array of bit flags. Values different from zero are interpreted as "bad" values and values equal to zero are considered as "good" values. However, see *ignore_bits* parameter on how to ignore some bits in the *bitmask* array.

ignore_bits : int, str, None

An integer bit mask, *None*, or a comma- or '+'-separated string list of integer bit values that indicate what bits in the input *bitmask* should be *ignored* (i.e., zeroed). If *ignore_bits* is a *str* and if it is prepended with '~', then the meaning of *ignore_bits* parameters will be reversed: now it will be interpreted as a list of bits to be *used* (or *not ignored*) when deciding what elements of the input *bitmask* array are "bad".

The *ignore_bits* parameter is the integer sum of all of the bit values from the input *bitmask* array that should be considered "good" when creating the output binary mask. For example, if values in the *bitmask* array can be combinations of 1, 2, 4, and 8 flags and one wants to consider that values having *only* bit flags 2 and/or 4 as being "good", then *ignore_bits* should be set to 2+4=6. Then a *bitmask* element having values 2,4, or 6 will be considered "good", while an element with a value, e.g., 1+2=3, 4+8=12, etc. will be interpreted as "bad".

Alternatively, one can enter a comma- or '+'-separated list of integer bit flags that should be added to obtain the final "good" bits. For example, both `4,8` and `4+8` are equivalent to setting *ignore_bits* to 12.

See *interpret_bits_value* for examples.

Setting *ignore_bits* to *None* effectively will interpret all *bitmask* elements as "good" regardless of their value.

Setting *ignore_bits* to 0 effectively will assume that all non-zero elements in the input *bitmask* array are to be interpreted as "bad".

In order to reverse the meaning of the *ignore_bits* parameter from indicating bits in the values of *bitmask* elements that should be ignored when deciding which elements are "good" (these are the elements that are zero after ignoring *ignore_bits*), to indicating the bits should be used exclusively in deciding whether a *bitmask* element is "good", prepend '~' to the string value. For example, in order to use **only** (or **exclusively**) flags 4 and 8 (2nd and 3rd bits) in the values of the input *bitmask* array when deciding whether or not that element is "good", set *ignore_bits* to `~4+8`, or `~4,8` To obtain the same effect with an `int` input value (except for 0), enter -(4+8+1)=-9. Following this convention, a `ignore_bits` string value of ``'~0'`` would be equivalent to setting `ignore_bits=None`.

good_mask_value : int, bool (Default = 1)

This parameter is used to derive the values that will be assigned to the elements in the output *mask* array that correspond to the "good" flags (that are 0 after zeroing bits specified by *ignore_bits*) in the input *bitmask* array. When *good_mask_value* is non-zero or *True* then values in the output mask array corresponding to "good" bit flags in *bitmask* will be 1 (or *True* if *dtype* is *bool*) and values of corresponding to "bad" flags will be 0. When *good_mask_value* is zero or *False* then values in the output mask array corresponding to "good" bit flags in *bitmask* will be 0 (or *False* if *dtype* is *bool*) and values of corresponding to "bad" flags will be 1.

dtype : data-type (Default = numpy.uint8)

The desired data-type for the output binary mask array.

Returns

mask : numpy.ndarray

Returns an array whose elements can have two possible values, e.g., 1 or 0 (or *True* or *False* if *dtype* is *bool*) according to values of to the input *bitmask* elements, *ignore_bits* parameter, and the *good_mask_value* parameter.

Examples

```
>>> from stsci.tools import bitmask
>>> dqbits = np.asarray([[0,0,1,2,0,8,12,0],[10,4,0,0,0,16,6,0]])
>>> bitmask.bitmask2mask(dqbits, ignore_bits=0, dtype=int)
array([[1, 1, 0, 0, 1, 0, 0, 1],
       [0, 0, 1, 1, 1, 0, 0, 1]])
>>> bitmask.bitmask2mask(dqbits, ignore_bits=0, dtype=bool)
array([[ True,  True, False, False,  True, False, False,  True],
       [False, False,  True,  True,  True, False, False,  True]], dtype=bool)
>>> bitmask.bitmask2mask(dqbits, ignore_bits=6, good_pix_value=0, dtype=int)
array([[0, 0, 1, 0, 0, 1, 1, 0],
       [1, 0, 0, 0, 0, 1, 0, 0]])
>>> bitmask.bitmask2mask(dqbits, ignore_bits=~6, good_pix_value=0, dtype=int)
array([[0, 0, 0, 1, 0, 0, 1, 0],
       [1, 1, 0, 0, 0, 0, 1, 0]])
>>> bitmask.bitmask2mask(dqbits, ignore_bits='~(2+4)', good_pix_value=0, dtype=int)
array([[0, 0, 0, 1, 0, 0, 1, 0],
       [1, 1, 0, 0, 0, 0, 1, 0]])
```

stsci.tools.bitmask.**interpret_bits_value**(*val*)

Converts input bits value from string to a single integer value or None. If a comma- or '+'-separated set of values are provided, they are summed.

Note: In order to flip the bits of the final result (after summation), for input of *str* type, prepend '~' to the input string. '~' must be prepended to the *entire string* and not to each bit flag!

Parameters

val : int, str, None

An integer bit mask or flag, *None*, or a comma- or '+'-separated string list of integer bit values. If *val* is a *str* and if it is prepended with '~', then the output bit mask will have its bits flipped (compared to simple sum of input val).

Returns

bitmask : int or None

Returns and integer bit mask formed from the input bit value or *None* if input *val* parameter is *None* or an empty string. If input string value was prepended with '~', then returned value will have its bits flipped (inverse mask).

Examples

```
>>> "{0:016b}".format(0xFFFF & interpret_bit_flags(28) )
'0000000000011100'
>>> "{0:016b}".format(0xFFFF & interpret_bit_flags('4,8,16') )
'0000000000011100'
>>> "{0:016b}".format(0xFFFF & interpret_bit_flags('~4,8,16') )
'1111111111100011'
>>> "{0:016b}".format(0xFFFF & interpret_bit_flags('~(4+8+16)') )
'1111111111100011'
```

CHAPTER

SIX

BUILDING A TEAL INTERFACE FOR TASKS

6.1 Cookbook for Building TEAL Interfaces

Abstract

The release of the Task Editor And Launcher(TEAL) with STScI_Python v2.10 in June 2010 provided the tools for building powerful GUI interfaces for editing the parameters of complex tasks and running those tasks with minimal effort. Learning how to use something new always takes a special effort, and this document provides a step-by-step walkthrough of how to build TEAL interfaces for any Python task to make this effort as easy as possible.

6.1.1 Introduction

The new TEAL GUI can be added to nearly any Python task that allows users to set parameters to control the operation of the task. Adding a TEAL interface to a Python task requires some minor updates to the task's code in order to allow TEAL to create and control the GUI for setting all the necessary parameters. TEAL itself relies on the ConfigObj module for the basic parameter handling functions, with additional commands for implementing enhanced logic for controlling the GUI itself based on parameter values. The GUI not only guides the user in setting the parameters, but also provides the capability to load and save parameter sets and the ability to read a help file while still editing the parameters. The interface to TEAL can also be set up alongside a command-line interface to the task. This document provides the basic information necessary for implementing a TEAL interface for nearly any Python task to take full advantage of the control it provides the user in setting the task parameters.

This document does not assume the user has any familiarity with using configobj in any manner and as as result includes very basic information which developers with some experience with configobj can simply skip over.

The development of the TEAL interface for the task *resetbits* in the *betadrizzle* package is used as an example. More elaborate examples will be explained after the development of the TEAL interface for *resetbits* has been described.

6.1.2 Building the Interface

The order of operations provided by this document is not the only order in which these steps can be performed. This order starts with the simplest operation then leads the developer into what needs to be done next with the least amount of iteration.

Step 1: Defining the Parameters

The primary purpose for developing a TEAL interface is to provide a GUI which can be used to set the values for the task's parameters. This requires that the developer identify the full set of task parameters which the user will be required to provide when running the task. The signature for the task *resetbits* is:

```
def reset_dq_bits(input,bits,extver=None,extname='dq')
```

These parameters now have to be described in a pair of configobj parameter files in order to define the parameters, their types and any validation that may need to be performed on the input values.

Default Values for the Parameters

The first file which needs to be defined provides the default values for each parameter. Default values can be any string or numerical value, including "" or None.

This task will simply need:

```
_task_name_ = resetbits
input = "*flt.fits"
bits = 4096
extver = None
extname = "dq"
```

The first line tells TEAL what task should be associated with this file. The default values for *extver* and *extname* simply match the defaults provided in the function signature. No default values were required for the other parameters, but these values were provided to support the most common usage of this task.

This file needs to be saved with a filename extension of *cfg* in a *pars/* subdirectory of the task's package. For *resetbits*, this file would be saved in the installation directory as the file:

```
betadrizzle/lib/pars/resetbits.cfg
```

This file will then get installed in the directory *betadrizzle/pars/resetbits.cfg* with the instructions on how to set that up coming in the last step of this process.

Parameter Validation Rules

The type for the parameter values, along with the definition of any range of valid values, is defined in the second configobj file: the configobj specification (configspec) file or *cfgspc* file. This file can also provide rules for how the GUI should respond to input values as well, turning the TEAL GUI into an active assistant for the user when editing large or complex sets of parameters.

For this example, we have a very basic set of parameters to define without any advance logic required. TEAL provides validators for a wide range of parameter types, including:

- ***strings*: matches any string**
  Defined using *string_kw()*
- ***integer*: matches any integer when a value is always required**
  Defined using *integer_kw()*
- ***integer* or *None*: matches any integer or a value of None**
  Defined using *integer_or_none_kw()*
- ***float*: matches any floating point value, when a value is always required**
  Defined using *float_kw()*
- ***float* or *None*: matches any floating point value or a value of None**
  Defined using *float_or_none_kw()*
- ***boolean*: matches boolean values - `True` or `False`**
  Defined using *boolean_kw()*
- ***option*: matches only those values provided in the list of valid options**
  Defined using *option_kw()* command with the list of valid values as a parameter

ConfigObj also has support for IP addresses as input parameters, and lists or tuples of any of these basic parameter types. Information on how to use those types, though, can be found within the ConfigObj module documentation.

With these available parameter types in mind, the parameters for the task can be defined in the configspec file. For the *resetbits* task, we would need:

```
_task_name_ = string_kw(default="resetbits")
input = string_kw(default="*flt.fits", comment="Input files (name, suffix, or @list)")
bits = integer_kw(default=4096, comment="Bit value in array to be reset to 0")
extver = integer_or_none_kw(default=None, comment="EXTVER for arrays to be reset")
extname = string_kw(default="dq", comment= "EXTNAME for arrays to be reset")
mode = string_kw(default="all")
```

Each of these parameter types includes a description of the parameter as the *comment* parameter, while default values can also be set using the *default* parameter value. This configspec file would then need to be saved alongside the .cfg file we just created as:

```
betadrizzle/lib/pars/resetbits.cfgspc
```

Note: If you find that you need or want to add logic to have the GUI respond to various parameter inputs, this can always be added later by updating the parameter definitions in this file. A more advanced example demonstrating how this can be done is provided in later sections.

Step 2: TEAL Functions for the Task

TEAL requires that a couple of functions be defined within the task in order for the GUI to know how to get the help for the task and to run the task. The functions that need to be defined are:

- **`run(configObj)`**
  This function serves as the hook to allow the GUI to run the task
- **`getHelpAsString()`**
  This function returns a long string which provides the help for the task

The sole input from TEAL will be a ConfigObj instance, a class which provides all the input parameters and their values after validation by the configobj validators. This instance gets passed by TEAL to the task's `run()` function and needs to be interpreted by that function in order to run the task.

Note: The `run()` and `getHelpAsString()` functions, along with the task's primary user interface function, all need to be in the module with the same name as the task, as TEAL finds the task by importing the taskname. Or, these two functions may instead be arranged as methods of a task class, if desired.

Defining the Help String

The help information presented by the TEAL GUI comes from the `getHelpAsString()` function and gets displayed in a simple ASCII window. The definition of this function can rely on help information included in the source code as docstrings or from an entirely separate file for tasks which have a large number of parameters or require long explanations to understand how to use the task. The example from the *resetbits* task was simply:

```
def getHelpAsString():
    helpString = 'resetbits Version '+__version__+__vdate__+'\n'
    helpString += __doc__+'\n'

    return helpString
```

This function simply relies on the module level docstring to describe how to use this task, since it is a simple enough task with only a small number of parameters.

Note: The formatting for the docstrings or help files read in by this function can use the numpy documentation restructured text markup format to be compatible with Sphinx when automatically generating documentation on this task using Sphinx. The numpy extension results in simple enough formatting that works well in the TEAL Help window without requiring any translation. More information on this format can be found in the Numpy Documentation pages.

More complex tasks may require the documentation which would be too long to comfortably fit within docstrings in the code itself. In those cases, separate files with extended discussions formatted using the numpy restructured text (reST) markup can be written and saved using the naming convention of `'<taskname>.help'` in the same directory as the module. The function can then simply use Python file operations to read it in as a list of strings which are concatenated together and passed along as the output. This operation has been made extremely simple through the definition of a new function within the TEAL package; namely, `teal.getHelpFileAsString()`. An example of how this could be used to extend the help file for *resetbits* would be:

```
def getHelpAsString():
    helpString = 'resetbits Version '+__version__+__vdate__+'\n'
    helpString += __doc__+'\n'
    helpString += teal.getHelpFileAsString(__taskname__,__file__)

    return helpString
```

The parameter __taskname__ should already have been defined for the task and gets used to find the file on disk with the name __taskname__.help. The parameter __file__ specifies where the task's module has been installed with the assumption that the help file has been installed in the same directory. The task *betadrizzle* uses separate files and can be used as an example of how this can be implemented.

Defining How to Run the Task

The ConfigObj instance passed by TEAL into the module needs to be interpreted and used to run the application. There are a couple of different models which can be used to define the interface between the run() function and the task's primary user interface function (i.e. how it would be called in a script).

1. The run() function interprets the ConfigObj instance and calls the user interface function. This works well for tasks which have a small number of parameters.
2. The run() function serves as the primary driver for the task and a separate function gets defined to provide a simpler interface for the user to call interactively. This works well for tasks which have a large number of parameters or sets of parameters defined in the ConfigObj interface.

Our simple example for the task resetbits uses the first model, since it only has the 4 parameters as input. The run() function can simply be defined in this case as:

```
def run(configobj=None):
    ''' Teal interface for running this code. '''

    reset_dq_bits(configobj['input'],configobj['bits'],
                  extver=configobj['extver'],extname=configobj['extname'])

def reset_dq_bits(input,bits,extver=None,extname='dq'):
```

Interactive use of this function would use the function reset_dq_bits(). The TEAL interface would pass the parameter values in through the run function to parse out the parameters and send it to that same function as if it were run interactively.

Step 3: Advertising TEAL-enabled Tasks

Any task which has a TEAL interface implemented can be advertised to users of the package through the use of a teal function: teal.print_tasknames(). This function call can be added to the package's *__init__.py* module so that everytime the package gets imported, or reloaded, interactively, it will print out a message listing all the tasks which have TEAL GUI's available for use. This listing will not be printed out when importing the package from another task. The *__init__.py* module for the *betadrizzle* package has the following lines:

```
# These lines allow TEAL to print out the names of TEAL-enabled tasks
# upon importing this package.
from stsci.tools import teal
teal.print_tasknames(__name__, os.path.dirname(__file__))
```

Step 4: Setting Up Installation

The additional files which have been added to the package with the task now need to be installed alongside the module for the task. Packages in the *STScI_Python* release get installed using Python's distutils mechanisms defined through the defsetup.py module. This file includes a dictionary for *setupargs* that describe the package and the files which need to be installed. This needs to be updated to include all the new files as data_files by adding the following line to the setupargs dictionary definition:

```
'data_files':  [(pkg+"/pars",['lib/pars/*']),( pkg, ['lib/*.help'])],
```

This will add the ConfigObj files in the *pars/* directory to the package while copying any `.help` files that were added to the same directory as the module.

Step 5: Testing the GUI

Upon installing the new code, the TEAL interface will be available for the task. There are a couple of ways of starting the GUI along with a way to grab the ConfigObj instance directly without starting up the GUI at all.

Running the GUI under PYRAF

The TEAL GUI can be started under PYRAF as if it were a standard IRAF task with the syntax:

```
>>> import <package>
>>> epar <taskname>
```

For example, our task `resetbits` was installed as part of the `betadrizzle` package, so we could start the GUI using:

```
>>> import betadrizzle
>>> epar resetbits
```

The fact that this task has a valid TEAL interface can be verified by insuring that the taskname gets printed out after the *import* statement.

Running the GUI using Python

Fundamentally, TEAL is a Python GUI that can be run interactively under any Python interpreter, not just PyRAF. It can be called for our example task using the syntax:

```
>>> from stsci.tools import teal
>>> cobj = teal.teal('resetbits')
```

Getting the ConfigObj Without Starting the GUI

The function for starting the TEAL GUI, `teal.teal()`, has a parameter to control whether or not to start the GUI at all. The ConfigObj instance can be returned for the task without starting the GUI by using the *loadOnly* parameter. For our example task, we would use the command:

```
>>> cobj = teal.teal('resetbits',loadOnly=True)
```

The output variable *cobj* can then be passed along or examined depending on what needs to be done at the time.

6.1.3 Advanced Topics

The topics presented here describe how to take advantage of some of TEAL's more advanced functions for controlling the behavior of the GUI and for working with complex sets of parameters.

Most of the examples for these advanced topics use the ConfgObj files and code defined for betadrizzle.

Parameter Sections

The ConfigObj specification allows for parameters to be organized into sections of related parameters. The parameters defined in these sections remain together in a single dictionary within the ConfigObj instance so that they can be passed into tasks or interpreted as a single unit. Use of sections within TEAL provides for the opportunity to control the GUI's behaviors based on whether or not the parameters in a given section need to be edited by the user.

A parameter section can be defined simply by providing a title using the following syntax in both the .cfg and .cfgspc files:

```
[<title>]
```

In betadrizzle, multiple sections are defined within the parameter interface. One section has been defined in the .cfg file as:

```
[STEP 1: STATIC MASK]
static = True
static_sig = 4.0
```

The .cfgspc definition for this section was specified as:

```
[STEP 1: STATIC MASK ]
static = boolean_kw(default=True, triggers='_section_switch_', comment="Create static bad-pixel mask
static_sig = float_kw(default=4.0, comment= "Sigma*rms below mode to clip for static mask")
```

These two sets of definitions work together to define the 'STEP 1: STATIC MASK' parameter section within the ConfigObj instance. A program can then access the parameters in that section using the name of the section as the index in the ConfigObj instance. The *static* and *static_sig* parameters would be accessed as:

```
>>> cobj = teal.teal('betadrizzle',loadOnly=True)
>>> step1 = cobj['STEP 1: STATIC MASK']
>>> step1
{'static': True, 'static_sig': 4.0}
>>> step1['static']
True
```

Section Triggers

The behavior of the TEAL GUI can be controlled for each section in a number of ways, primarily as variations on the behavior of turning off the ability to edit the parameters in a section based on another parameters value. A section parameter can be defined to allow the user to explicitly specify whether or not they need to work with those parameters. This can the control whether or not the remainder of the parameters are editable through the use of the *triggers* argument in the .cfgspc file for the section parameter.

The supported values for the *triggers* argument currently understood by TEAL are:

- `_section_switch_`: Activates/Deactivates the ability to edit the values of the parameters in this section
- `_rule<#>_`: Runs the code in this rule (defined elsewhere in the .cfgspc file) to automatically set this parameter, and control the behavior of other parameters like section defintions as well.

The example for defining the section 'STEP 1: STATIC MASK' illustrates how to use the `_section_switch_` trigger to control the editing of the parameters in that section.

Another argument defined as `is_set_by="_rule<#>"` allows the user to define when this section trigger can be set by other parameters using code and logic provided by the user. The value, `_rule<#>_` refers to code in the specified rule (defined at the end of the *cfgspc* file) to determine what to do. The code which will be run must be found in the configspec file itself, although that code could reference other packages which are already installed.

Use of Rules

A special section can be appended to the end of the ConfigObj files (.cfg and .cfgspc files) to define rules which can implement nearly arbitrary code to determine how the GUI should treat parameter sections or even individual parameter settings. The return value for a rule should always be a boolean value that can be used in the logic of setting parameter values.

This capability has been implemented in *betadrizzle* to control whether or not whole sections of parameters are even editable (used) to safeguard the user from performing steps which need more than 1 input when only 1 input is provided. The use of the `_rule<#>_` trigger can be seen in the *betadrizzle* .cfgspc file:

```
_task_name_ = string_kw(default="betadrizzle")
input = string_kw(default="*flt.fits", triggers='_rule1_', comment="Input files (name, suffix, or @li

<other parameters removed...>

[STEP 3: DRIZZLE SEPARATE IMAGES]
driz_separate = boolean_kw(default=True, triggers='_section_switch_', is_set_by='_rule1_', comment= '
driz_sep_outnx = float_or_none_kw(default=None, comment="Size of separate output frame's X-axis (pixe

<more parameters removed, until we get to the end of the file...>

[ _RULES_ ]
_rule1_ = string_kw(default='', when='defaults,entry', code='from stsci.tools import check_files; ans
```

In this case, `_rule1_` gets defined in the special parameter section `[_RULES_]` and triggered upon the editing of the parameter `input`. The result of this logic will then automatically set the value of any section parameter with the `is_set_by=_rule1_` argument, such as the parameter `driz_separate` in the section `[STEP 3: DRIZZLE SEPARATE IMAGES]`

The rule is executed within Python via two reserved words: `VAL`, and `OUT`. `VAL` is automatically set to the value of the parameter which was used to trigger the execution of the rule, right before the rule is executed. `OUT` will be the outcome of the rule code - the way it returns data to the rule execution machinery without calling a Python *return*.

For the rule itself, one can optionally state (via the `when` argument) when the rule will be evaluated. The currently supported options for the `when` argument (used for rules only) are:

- `init`: Evaluate the rule upon starting the GUI
- `defaults`: Evaluate the rule when the parameter value changes because the user clicked the "Defaults" button
- `entry`: Evaluate the rule any time the value is changed in the GUI by the user manually
- `fopen`: Evaluate the rule any time a saved file is opened by the user, changing the value
- `always`: Evaluate the rule under any of these circumstances

These options can be provided as a comma-separated list for combinations, although care should be taken to avoid any logic problems for when the rule gets evaluated. If a `when` argument is not supplied, the value of `always` is assumed.

Tricky Rules

A parameter can also be controlled by multiple other parameters using the same rule. The example below shows how to get `par1` to be grayed out if `do_step1` and `do_step2` are both disabled.

In the .cfgspc file:

```
_task_name_ = string_kw(default="mytask")
par1 = string_kw(default="", active_if="_rule1_", comment="Shared parameter")

<other parameters removed...>

[STEP 1: FOO]
do_step1 = boolean_kw(default=True, triggers='_section_switch_', triggers='_rule1_', comment="Do Step

<other parameters removed...>

[STEP 2: BAR]
do_step2 = boolean_kw(default=True, triggers='_section_switch_', triggers='_rule1_', comment="Do Step
```

```
<more parameters removed, until we get to the end of the file...>

[ _RULES_ ]

_rule1_ = string_kw(default='', code='import mytask; OUT = mytask.tricky_rule(NAME, VAL)')
```

In mytask.py file:

```
MY_FLAGS = {'do_step1': 'yes', 'do_step2': 'yes'}

def tricky_rule(in_name, in_val):
    global MY_FLAGS
    MY_FLAGS[in_name] = in_val
    if MY_FLAGS['do_step1'] == 'yes' or MY_FLAGS['do_step2'] == 'yes':
        ans = True
    else:
        ans = False
    return ans
```

For the rule itself, each rule has access to:

- `SCOPE`
- `NAME` - Parameter name.
- `VAL` - Parameter value.
- `TEAL` - Reference to the main TEAL object, which knows the value of all of its parameters. However, `TEAL.getValue(NAME)` returns its value *before* it is updated.

To debug your tricky rule, you can add print-out lines to your rule. TEAL log under `Help` menu also shows you what it is doing.

CHAPTER

SEVEN

INDICES AND TABLES

- genindex
- modindex
- search

S

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

X