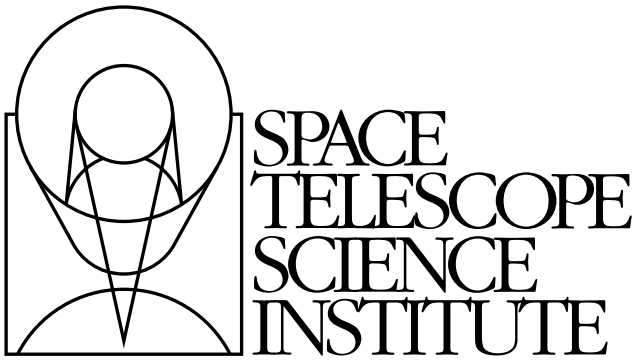

stsci.tools Documentation

Release 2.9

SSB

May 03, 2013

CONTENTS

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 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*)
Built 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.

Usage:
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*)
This function 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. It 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: a tuple (isfits, fitstype) :

The values of *isfits* and *fitstype* are specified as:

```
isfits - True|False
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).

For files with 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. It will work 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 pyfits 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 pyfits module which offers additional features specific to STScI. These features include the handling of Constant Data Value Arrays.

The pyfits module is:

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 pyfits.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

convertwaiveredfits: Convert a waivered FITS file to various other formats.

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

Syntax From the command line
convertwaiveredfits.py [-hm] [-o <outputFileName>,...] FILE ...

Convert the waivered FITS files (FILEs) to various formats. The default conversion format is multi-extension FITS.

Options

-h, --help	print this help message and exit
-v, --verbose	turn on verbose output
-m, --multiExtensionConversion	convert to multi-extension FITS format (Default)
-o, --outputFileName	comma separated list of output file specifications one per input FILE Default: input file specification with the last character of the base name changed to *h* in multi-extension FITS format

Examples

Conversion of a WFPC2 waivered FITS file obtained from the HST archive:

```
convertwaiveredfits u9zh010bm_c0f.fits
```

This will convert the waivered FITS file *u9zh010bm_c0f.fits* to multi-extension FITS format and generate the output file *u9zh010bm_c0h.fits*.

Conversion of multiple FITS files can be done using:

```
convertwaiveredfits -o out1.fits,out2.fits
                    u9zh010bm_c0f.fits u9zh010bm_c1f.fits
```

This will convert the waivered FITS files u9zh010bm_c0f.fits and u9zh010bm_c1f.fits to multi-extension FITS format and generate the output files out1.fits and out2.fits

Python Syntax

You can run this code interactively from within Python using the syntax:

```
>>> from stsci.tools import convertwaiveredfits
>>> fobj = convertwaiveredfits.convertwaiveredfits(waiveredObject,
>>>                     outputFileName=None,
>>>                     forceFileOutput=False,
>>>                     convertTo='multiExtension',
>>>                     verbose=False)
```

The returned object *fobj* is a PyFITS object using the multi-extension FITS format.

Parameters

waiveredObject: obj
input object representing a waivered FITS file; either a pyfits.HDUList object, a file object, or a file specification

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

forceFileOutput: boolean
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: string
target conversion type Default: 'multiExtension'

verbose: boolean
provide verbose output Default: False

Returns

hduList
pyfits.HDUList (PyFITS multi-extension FITS object) containing converted output

Examples

```
>>> import convertwaiveredfits
>>> hdulist = convertwaiveredfits.convertwaiveredfits('u9zh010bm_c0f.fits',
                                    forceFileOutput=True)
```

this will convert the waivered FITS file u9zh010bm_c0f.fits to multi-extension FITS format and write the output to the file u9zh010bm_c0h.fits; the returned HDUList is in multi-extension FITS format

```
>>> import convertwaiveredfits
>>> inFile = open('u9zh010bm_c0f.fits',mode='rb')
>>> hdulist = convertwaiveredfits.convertwaiveredfits(inFile,
                                                      'out.fits')
```

this will convert the waivered FITS file u9zh010bm_c0f.fits to multi-extension FITS format and write the output to the file out.fits; the returned HDUList is in multi-extension FITS format

```
>>> import pyfits
>>> import convertwaiveredfits
>>> inHdul = pyfits.open('u9zh010bm_c0f.fits')
>>> hdulist = convertwaiveredfits.convertwaiveredfits(inHdul)
```

this will convert the waivered FITS file u9zh010bm_c0f.fits to multi-extension FITS format; no output file is generated; the returned HDUList is in multi-extension format

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 pyfits.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 pyfits.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*)

Notes

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*)

Notes

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': a list :

a python list of filenames

'output' a string :

a user specified output name or 'final'

'shiftfile': a string :

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*)

Purpose

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': string :

name of shift file on disk, see above the expected format

'form': string :

form of shifts (absolute|delta)

'frame': string :

frame in which the shifts should be applied (input|output)

'units': string :

in which the shofts are measured (pixels?)

'order': list :

Keeps track of the order of the files

'refimage': string :

name of reference image

'd': dictionary** :

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.

1.Read a shift file on disk using:

```
>>> sdict = ShiftFile('shifts.txt')
```

2.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=['j8bt06ny
 '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.

Algorithm
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': string :

name of association table

'output': string :

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-object :

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

BUILDING A TEAL INTERFACE FOR TASKS

5.1 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.

5.1.1 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.

5.1.2 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.

5.1.3 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.

CHAPTER

SIX

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

S

S

W

X