Streamlining an IRAF data reduction process with AstroPy and NDMapper

James E. H. Turner

Background

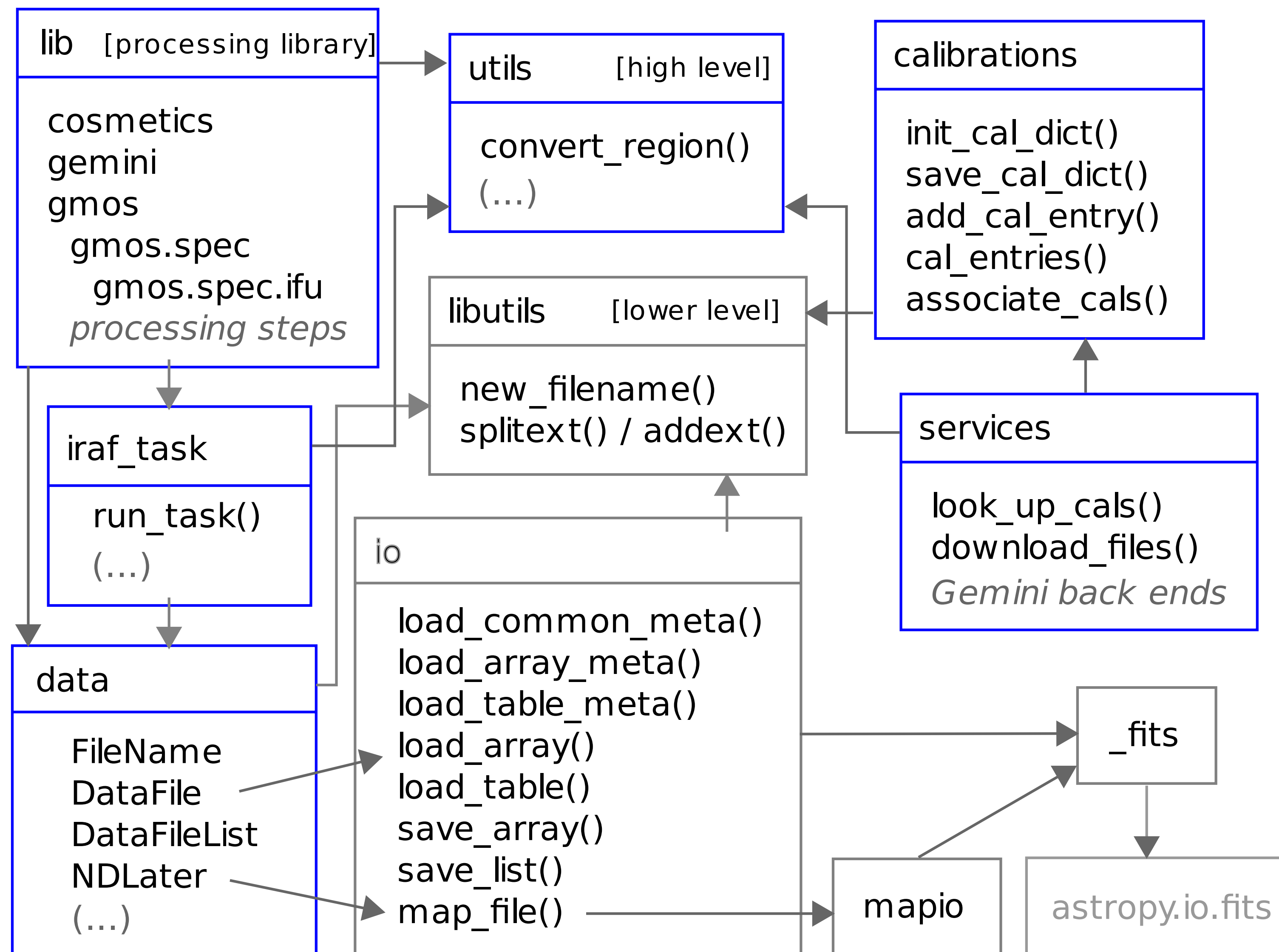
- Working with a larger number of GMOS IFU observations (optical) than in past science projects.
- Have done recent work on the quality of the IRAF process but it's still very time consuming to work with & adapt for new data in CL, eg.:
    - Filenames repeated numerous times throughout the DR script.
        - Process not separated from the data generally -- brittle.
    - Inconsistent support for passing lists of files to different steps.
    - Re-processing means deleting files, commenting bits out etc.
        - Easy to make a mistake but costly.
            - Full process takes ~6hrs.
        - Script can end up not reflecting what was actually done.
    - Calibrations have to be selected & specified by user for each step.
    - No global settings for interactivity, error propagation etc.
    - Lots of routine boilerplate changes to parameter defaults.

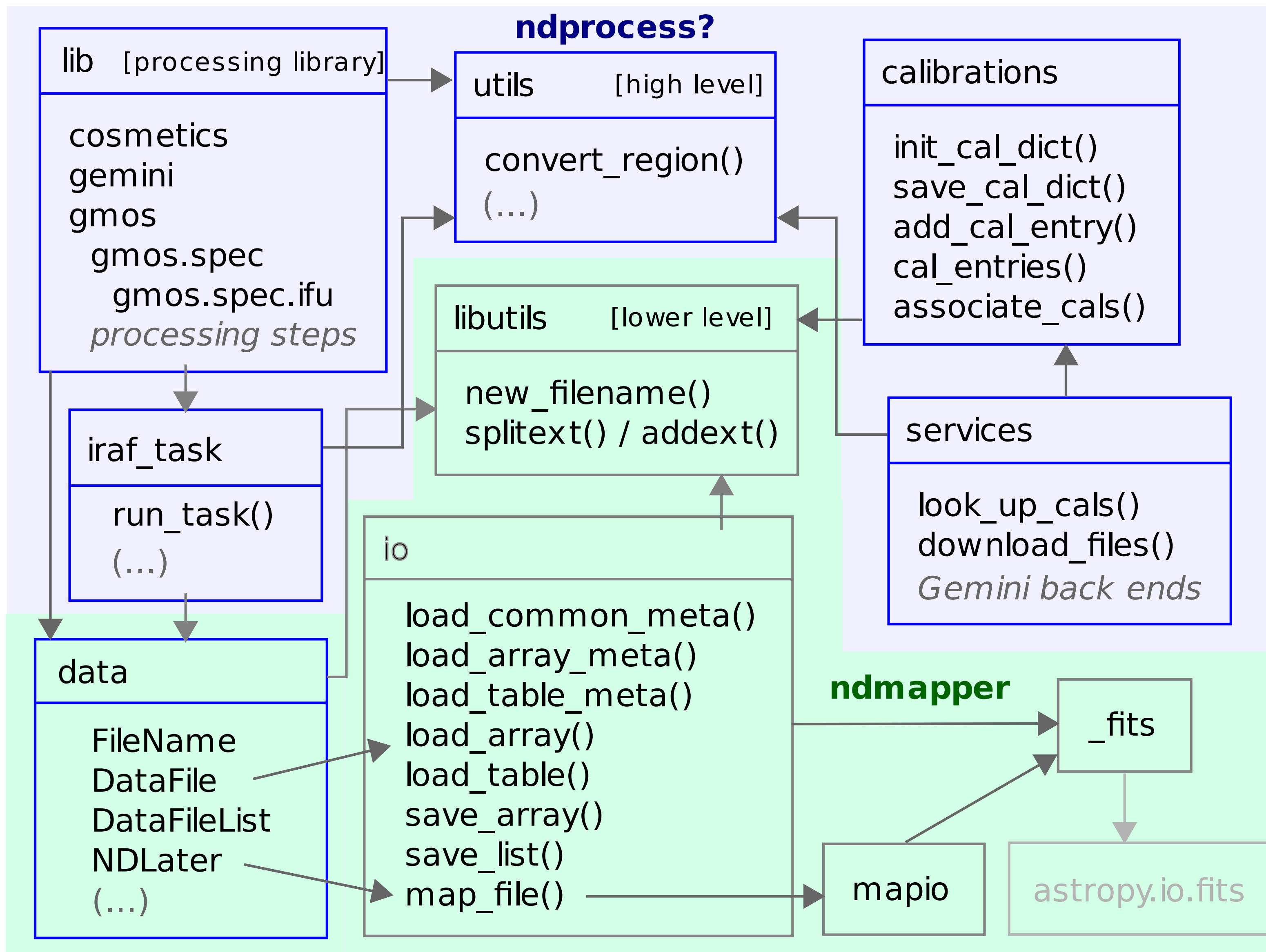
Background (contd.)

- Eg. (contd.):
  - General verbosity (eg, 1000 lines) obfuscates the process.
  - Some rather monolithic processing steps.
  - Obscure methods for BPM creation & bug work-arounds.
  - http://drforum.gemini.edu/wp-content/uploads/2015/04/allred.cl (http://drforum.gemini.edu/wp-content/uploads/2015/04/allred.cl).
- Also:
  - Want ability to insert new Python steps.
  - Want to interoperate with AstroPy.
- Opt to re-write the processing sequence in Python, instead of spending time on monkey work.
  - Best option to use *existing* reduction code more efficiently with reasonable development effort/risk & Python compatibility was to start something new.
  - Emphasize convenience & productivity.

Outline

Support code to allow scripting the IRAF process simply in Python.




ndprocess?
lib [processing library]
cosmetics
gemini
gmos
gmos.spec
gmos.spec.ifu
processing steps
utils [high level]
convert_region()
(...)
calibrations
init_cal_dict()
save_cal_dict()
add_cal_entry()
cal_entries()
associate_cals()
libutils [lower level]
new_filename()
splitext() / addext()
iraf_task
run_task()
(...)
services
look_up_cals()
download_files()
Gemini back ends
io
load_common_meta()
load_array_meta()
load_table_meta()
load_array()
load_table()
save_array()
save_list()
map_file()
data
FileName
DataFile
DataFileList
NDLater
(...)
ndmapper
_fits
mapio
astropy.io.fits

Data representation

In [1]:
```
# Use an example that looks like this in FITS:

from astropy.io import fits

fits.info('ergS20120827S0066.fits')
```

```
Filename: ergS20120827S0066.fits
No.    Name         Type      Cards   Dimensions   Format
0    PRIMARY     PrimaryHDU     251   ()
1    MDF         BinTableHDU     43   1500R x 8C   [1J, 1E, 1E, 5A, 1J, 1J, 1J, 1J]
2    SCI         ImageHDU      1583   (2879, 743)   float32
3    VAR         ImageHDU      1583   (2879, 743)   float32
4    DQ          ImageHDU      1583   (2879, 743)   int16 (rescales to uint16)
5    SCI         ImageHDU      1583   (2879, 745)   float32
6    VAR         ImageHDU      1583   (2879, 745)   float32
7    DQ          ImageHDU      1583   (2879, 745)   int16 (rescales to uint16)
```

In [2]:
```
from ndmapper.data import *

# Open the FITS file as a list of NDData instances:
df = DataFile('ergS20120827S0066.fits')
df
```

Out[2]:
```
DataFile 'ergS20120827S0066.fits' (len 2)
```

In [3]:
```
# The FITS primary header lives in a file-level `meta` attribute:
df.meta[:10]  # print first 10 lines
```

Out[3]:
```
SIMPLE  =                    T / Fits standard
BITPIX  =                   16 / Bits per pixel
NAXIS   =                    0 / Number of axes
EXTEND  =                    T / File may contain extensions
ORIGIN  = 'NOAO-IRAF FITS Image Kernel July 2003' / FITS file originator
DATE    = '2016-01-22T23:43:30' / Date FITS file was generated
IRAF-TLM= '2016-01-29T22:21:56' / Time of last modification
COMMENT   FITS (Flexible Image Transport System) format is defined in 'Astronomy
COMMENT   and Astrophysics', volume 376, page 359; bibcode: 2001A&A...376..359H
INSTRUME= 'GMOS-S  '           / Instrument used to acquire data
```

In [4]:
```
# The DataFile behaves as a list of NDDataArray instances, with the SCI, VAR & DQ FITS
# extensions translated to data, uncertainty & flags, respectively:
df[0].data   # SCI
```

Out[4]:
```
array([[ 16.32850838,  37.23521423,  50.67963028, ...,   0.        ,
          0.        ,   0.        ],
       [ 23.59534264,  14.43008041,  45.55400467, ...,   0.        ,
          0.        ,   0.        ],
       [  0.        ,   0.        ,  23.65302849, ...,   0.        ,
          0.        ,   0.        ],
       ...,
       [ 64.25991821,  48.29379272,  35.84579468, ...,   0.        ,
          0.        ,   0.        ],
       [ 75.59520721,  40.56682205,  18.23557091, ...,   0.        ,
          0.        ,   0.        ],
       [ 79.64089966,  61.27500153,  83.26106262, ...,   0.        ,
          0.        ,   0.        ]], dtype=float32)
```

In [5]:
```
df[0].uncertainty  # VAR
```

Out[5]:
```
<astropy.nddata.nduncertainty.StdDevUncertainty at 0x7f1576990d10>
```

In [6]:
```
df[0].flags  # DQ
```

Out[6]:
```
array([[0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0],
       ...,
       [0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0]], dtype=uint16)
```

In [7]:
```
df[0].meta[:10]
```

Out[7]:
```
XTENSION= 'IMAGE   '           / Image extension
BITPIX  =                  -32 / Bits per pixel
NAXIS   =                    2 / Number of axes
NAXIS1  =                 2879 / Axis length
NAXIS2  =                  743 / Axis length
PCOUNT  =                    0 / No 'random' parameters
GCOUNT  =                    1 / Only one group
EXTNAME = 'SCI     '           / Extension name
EXTVER  =                    1 / Extension version
INHERIT =                    F / Inherits global header
```

In [8]:
```
# The name is kept in a filename parser object:
# (which uses a configurable regexp to recognize the base name)
df.filename
```

Out[8]:
```
FileName 'ergS20120827S0066.fits'
```

In [9]:
```
df.filename.dir, df.filename.prefix, df.filename.base, df.filename.ext  # etc.
```

Out[9]:
```
('', 'erg', 'S20120827S0066', 'fits')
```

In [10]:
```
# Convert to a new name with modifiers:
FileName(df.filename, prefix='t', suffix='_sum', dirname='')
```

Out[10]:
```
FileName 'tergS20120827S0066_sum.fits'
```

In [11]:
```
df.filename == '../talk/ergS20120827S0066.fits'   # path equivalence
```

Out[11]:
```
True
```

In [12]:
```
str(df)
```

Out[12]:
```
'ergS20120827S0066.fits'
```

New instance

In [13]:
```
from astropy.nddata import NDData

df=DataFile('arthur.fits', mode='overwrite')  # 'read', 'update', 'new' or 'overwrite'
df.append(NDData([1,2,3,4]))
df.append(NDData([5,6,7]))
df
```

Out[13]:
```
DataFile 'arthur.fits' (len 2)
```

In [14]:
```
df[0].ident  # FITS EXTVER; can be changed
```

Out[14]:
```
1
```

In [15]:
```
df.save()
```

In [16]:
```
fits.info('arthur.fits')
```

```
Filename: arthur.fits
No.    Name         Type      Cards   Dimensions   Format
0    PRIMARY     PrimaryHDU       4   ()
1    SCI         ImageHDU         9   (4,)         int64
2    SCI         ImageHDU         9   (3,)         int64
```

- FITS I/O is abstracted into a replaceable back end, but ...
  - Files must map to a flat list of `NDData` instances.
  - Currently `ident` must be integer.
- Binary tables get propagated as AstroPy Table instances.
  - Preserves information on loading & saving.
  - But no public interface defined for accessing them yet.
- Unrecognized image extensions are ignored for now (expect access via something like "extras" attribute).
- Pixel data are lazily loaded, avoiding duplicate memory use when just running an IRAF task etc.
  - Achieved using a sub-class of `NDDataArray`, shamelessly called `NDLater`.
  - Instantiated with a proxy object that requests data, uncertainty & flags from `io.fits` on demand.

File lists

In [17]:

```
# Lists of DataFile objects can be instantiated using a DataFileList object:

dfl = DataFileList(['S20120827S0066.fits', 'S20120827S0067.fits', \
                    'S20120827S0068.fits'], prefix='erg')
dfl.append(df)
dfl
```

Out[17]:

```
[DataFile 'ergS20120827S0066.fits' (len 2),
 DataFile 'ergS20120827S0067.fits' (len 2),
 DataFile 'ergS20120827S0068.fits' (len 2),
 DataFile 'arthur.fits' (len 2)]
```

Serves dual purpose of tracking lists of filenames & accessing the data.

Functionality currently minimal but envisage ops. on multiple `DataFile` objects.

Interfacing with IRAF

In [18]:

```
from ndmapper.iraf_task import run_task

dfl1 = DataFileList(['ergS20120827S0066.fits', 'ergS20120827S0067.fits',
                     'ergS20120827S0068.fits'])
dfl2 = dfl1[::-1]  # flip it for fun

dfout1 = run_task('gemini.gemtools.gemarith', \
                  inputs={'operand1' : dfl1, 'operand2' : dfl2}, \
                  outputs={'result' : '@operand1'}, \
                  suffix='_test1', MEF_ext=False, op='+')
```

```
-----
START run_task(gemini.gemtools.gemarith)  2016-03-24 22:38:38
GEMARITH begin (this task relies on gemexpr)
GEMARITH PHU copy: ergS20120827S0066.fits[0] --> ergS20120827S0066_test1.fits[0,APPEND])
GEMARITH Going to copy MDF from ergS20120827S0066.fits[1]
GEMARITH gemarith: done
GEMARITH begin (this task relies on gemexpr)
GEMARITH PHU copy: ergS20120827S0067.fits[0] --> ergS20120827S0067_test1.fits[0,APPEND])
GEMARITH Going to copy MDF from ergS20120827S0067.fits[1]
GEMARITH gemarith: done
GEMARITH begin (this task relies on gemexpr)
GEMARITH PHU copy: ergS20120827S0068.fits[0] --> ergS20120827S0068_test1.fits[0,APPEND])
GEMARITH Going to copy MDF from ergS20120827S0068.fits[1]
GEMARITH gemarith: done
END   run_task(gemini.gemtools.gemarith)  2016-03-24 22:38:39
-----
```

```
In [19]: dfout1
```

```
Out[19]: {'result': [DataFile 'ergS20120827S0066_test1.fits' (len 2),
           DataFile 'ergS20120827S0067_test1.fits' (len 2),
           DataFile 'ergS20120827S0068_test1.fits' (len 2)]}
```

Can loop over files and FITS extensions, handling several bookkeeping tasks.

Input DataFile instances get auto-saved for IRAF if they might have changed in memory.

New processing library API etc.

- Thin wrappers for IRAF tasks, using `run_task()`.
  - Simplify the old API & set lots of parameter defaults.
  - Obey global settings, eg. for interactivity.
    - A standard decorator takes care of the bookkeeping.
  - Look virtually the same as pure Python steps (with docstrings).
- Currently have 2 simple proof-of-concept steps in pure Python.
  - Also fairly easy to define.
    - Determine file names & modes, loop over files & `NDData` instances and do the appropriate calculation.
    - Simplify further with a meta-decorator for iteration etc?
- Option to overwrite or re-use existing files automatically.
- Hierarchical instrument library exposes inherited processing steps or overloaded, mode-specific versions as needed.
  - Or user can substitute their modified version in a later import.

Managing files & calibration matches

- Modules `calibrations` and `services` provide automated calibration matching & file downloads (currently for Gemini).
- Calibrations matched recursively to get everything needed up front.
  - Results cached in a user-editable JSON file.
    - Easy to override what's used.
  - Dependence on calibration types defined by a modifiable dict, imported from the instrument library.
  - Matches determined by external Web service for Gemini data.
- Both file download & calibration matching steps become no-ops when previously completed.
  - Reprocessing is reproducible & can be done off line.
- Support functions allow 1-line operations.
  - Eg. iterating over dict entries for a calibration type.
- Checksums verified automatically when downloading; no bad copies.
- Only ~15% of code in Gemini-specific back end.

```python
# A draft Python version of my GMOS IFU data reduction example,
# by James E.H. Turner, Jan. 2016.

import ndmapper
from ndmapper.data import FileName, DataFile, DataFileList
from ndmapper.services import look_up_cals, download_files
from ndmapper.calibrations import cal_entries, associate_cals

from ndmapper.lib.cosmetics import init_bpm, add_bpm
from ndmapper.lib.gmos.spec.ifu import *

ndmapper.config['logfile'] = 'example.log'
ndmapper.config['reprocess'] = False
ndmapper.config['interact'] = False

rawdir = 'raw'


# Download specified raw science exposures & open them as DataFiles:
obj_spec = DataFileList(
    download_files(['S20120827S0066.fits', 'S20120827S0067.fits',
                    'S20120827S0068.fits'], server='gemini', dirname=rawdir)
)

# Look up the corresponding calibration files (& their calibrations in turn)
# in the Gemini archive. For now, a "trace" entry has to be added manually for
# the arc after running this ("trace": "S20120827S0069_flat.fits") but is
# preserved when re-running the step.
cal_dict = look_up_cals(obj_spec, dependencies=CAL_DEPS, server='gemini',
                        cache='calibrations.json')

# Download the matching calibrations:
download_files(cal_dict, server='gemini', dirname=rawdir)


# Process the biases:
for name, flist in cal_entries(cal_dict, 'bias'):

    bias_im = DataFileList(flist, dirname=rawdir)
```

Try it yourself!

Installation from PyPI (v0.1.1 from 12 Mar.) -- requires PyRAF:

```
# Ureka environment:
ur_setup -n ndmapper_test

# Conda (Python 2) environment:
conda create --name ndmapper_test astropy
source activate ndmapper_test

# Install:
pip install ndmapper
```

API documentation: http://ndmapper.readthedocs.org (http://ndmapper.readthedocs.org)

Source code: https://github.com/jehturner/ndmapper (https://github.com/jehturner/ndmapper)

(A few library steps require minor fixes from my Gemini IRAF version at http://drforum.gemini.edu/topic/gmos-ifu-data-reduction-scripts/ (http://drforum.gemini.edu/topic/gmos-ifu-data-reduction-scripts/).)