reduce Users Manual

Release X1.0.1

Kenneth Anderson

October 07, 2016

CONTENTS

CHAPTER

ONE

INTRODUCTION

This document is version 1.0 of the `reduce` Users Manual. This manual will describe the usage of `reduce` as an application provided by the Gemini Observatory Astrodata package suite. `reduce` is an application that allows users to invoke the Gemini Recipe System to perform data processing and reduction on one or more astronomical datasets.

This document presents details on applying `reduce` to astronomical datasets, currently defined as multi-extension FITS (MEF) files, both through the application's command line interface and the application programming interface (API). Details and information about the `astrodata` package, the Recipe System, and/or the data processing involved in data reduction are beyond the scope of this document and will only be engaged when directly pertinent to the operations of `reduce`.

1.1 Reference Documents

- *The Gemini Recipe System: a dynamic workflow for automated data reduction*, K. Labrie *et al*, SPIE, 2010.
- *Developing for Gemini's extensible pipeline environment*, K. Labrie, C. Allen, P. Hirst, ADASS, 2011
- *Gemini's Recipe System; A publicly available instrument-agnostic pipeline infrastructure*, K. Labrie et al, ADASS 2013.

1.2 Overview

As an application, `reduce` provides interfaces to configure and launch the Gemini Recipe System, a framework for developing and running configurable data processing pipelines and which can accommodate processing pipelines for arbitrary dataset types. In conjunction with the development of `astrodata`, Gemini Observatory has also developed the `gemini_instruments` and `GeminiDR` packages, the code base currently providing abstraction of, and processing for, Gemini Observatory astronomical observations.

In Gemini Observatory's operational environment "on summit," `reduce`, `astrodata`, and the `gemini_instruments` packages provide a currently defined, near-realtime, quality assurance pipeline, the so-called QAP. `reduce` is used to launch this pipeline on newly acquired data and provide image quality metrics to observers, who then assess the metrics and apply observational decisions on telescope operations.

Users unfamiliar with terms and concepts heretofore presented should consult documentation cited in the previous sections (working on the Recipe System User Manual).

1.3 Glossary

adcc – Automatated Data Communication Center. Provides HTTP service for moinitoring QA metrics produced during pipeline operations. This is run externally to `reduce`. Users need not know about or

invoke the `adcc` for `reduce` operations.

astrodata (or Astrodata) – part of the **gemini_python** package suite that defines the dataset abstraction layer for the Recipe System.

AstroData – not to be confused with **astrodata**, this is the main class of the `astrodata` package, and the one most users and developers will interact with at a programmatic level.

AstroData tags – Astrodata tags Represents a data classification. A dataset will be classified by a number of types that describe both the data and its processing state. For example, a typical unprocessed GMOS image would have a set of tags like

set(['RAW', 'GMOS', 'GEMINI', 'SIDEREAL', 'UNPREPARED', 'IMAGE', 'SOUTH']) (see **tags** below).

Descriptor – Represents a high-level metadata name. Descriptors allow access to essential information about the data through a uniform, instrument-agnostic interface to the FITS headers.

gemini_python – A suite of packages comprising **astrodata**, **gemini_instruments**, the **recipe system** and **gempy**, all of which provide the full functionality needed to run recipe pipelines on observational datasets.

gempy – a **gemini_python** package comprising gemini specific functional utilities.

MEF – Multiple Extension FITS, the standard data format not only for Gemini Observatory but many observatories.

primitive – A function defined within an **GeminiDR** package that performs actual work on the passed dataset. Primitives observe tightly controlled interfaces in support of re-use of primitives and recipes for different types of data, when possible. For example, all primitives called `flatCorrect` must apply the flat field correction appropriate for the data's current astrdata tag set, and must have the same set of input parameters. This is a Gemini Coding Standard, it is not enforced by the Recipe System.

recipe – Represents the sequence of transformations, which are defined as methods on a primitive class. A recipe is a simple python function recieves an instance of the the appropriate primitive class and calls the available methods that are to be done for a given recipe function. A **recipe** is the high-level pipeline definition. Users can pass recipe names directly to reduce. Essentially, a recipe is a pipeline.

Recipe System – The gemin_python framework that accommodates an arbitrary number of defined recipes and the primitives

reduce – The command line interface to the Recipe System and its associated recipes/pipelines.

tags or **tag set** – these are tags that characterise the dataset and defined in a `gemini_instruments` instrument package used to describe the kind of observational data that has been passed to the Recipe System., Eg., a GMOS IMAGE; a NIRI IMAGE.

CHAPTER

TWO

INSTALLATION

The `astrodata` package has several dependencies like `numpy`, `astropy`, and others. All dependencies of `gemini_python` and `astrodata` are provide by the Ureka package, and users are highly encouraged to install and use this very useful package. It is an easy and, perhaps, best way to get everything you need and then some. Ureka is available at http://ssb.stsci.edu/ureka/.

WARNING: The Ureka installation script will not set up IRAF for you. You need to do that yourself. Here's how:

```
$ cd ~
$ mkdir iraf
$ cd iraf
$ mkiraf
-- creating a new uparm directory
Terminal types: xgterm,xterm,gterm,vt640,vt100,etc.
Enter terminal type: xgterm
A new LOGIN.CL file has been created in the current directory.
You may wish to review and edit this file to change the defaults.
```

Once a user has has retrieved the gemini_python package, available as a tarfile from the Gemini website (http://gemini.edu), and untarred only minor adjustments need to be made to the user environment in order to make astrodata importable and allow `reduce` to work properly.

2.1 Install

2.1.1 Recommended Installation

It is recommended to install the software in a location other than the standard python location for modules (the default `site-packages`). This is also the only solution if you do not have write permission to the default `site-packages`. Here is how you install the software somewhere other than the default location:

```
$ python setup.py install --prefix=/your/favorite/location
```

`/your/favorite/location` must already exist. This command will install executable scripts in a `bin` subdirectory, the documentation in a `share` subdirectory, and the modules in a `lib/python2.7/site-packages` subdirectory. The modules being installed are `astrodata`, `astrodata_FITS`, `astrodata_Gemini`, and `gempy`. In this manual, we will only use `astrodata`.

Because you are not using the default location, you will need to add two paths to your environment. You might want to add the following to your .cshrc or .bash_profile, or equivalent shell configuration script.

C shell(csh, tcsh):

```
setenv PATH /your/favorite/location/bin:${PATH}
setenv PYTHONPATH /your/favorite/location/lib/python2.7/site-packages:${PYTHONPATH}
```

Bourne shells (sh, bash, ksh, ...)

```
export PATH=/your/favorite/location/bin:${PATH}
export PYTHONPATH=/your/favorite/location/lib/python2.7/site-packages:${PYTHONPATH}
```

If you added those lines to your shell configuration script, make sure your `source` the file to activate the new setting.

For csh/tcsh:

```
$ source ~/.cshrc
$ rehash
```

For bash:

```
$ source ~/.bash_profile
```

2.1.2 Installation under Ureka

Assuming that you have installed Ureka and that you have write access to the Ureka directory, this will install `astrodata` in the Ureka `site-packages` directory. WARNING: While easier to install and configure, this will modify your Ureka installation.

```
$ python setup.py install
```

This will also add executables to the Ureka `bin` directory and documentation to the Ureka `share` directory.

With this installation scheme, there is no need to add paths to your environment. However, it is a lot more complicated to remove the Gemini software in case of problems, or if you just want to clean it out after evaluation.

In tcsh, you will need to run `rehash` to pick the new executables written to `bin`.

2.2 Test the installation

Start up the python interpreter and import astrodata:

```
$ python
>>> import astrodata
```

Next, return to the command line and test that `reduce` is reachable and runs. There may be some delay as package modules are byte compiled:

```
$ reduce -h
```

or

```
$ reduce [--help]
```

This will print the reduce help to the screen.

If users have Gemini fits files available, they can test that the Recipe System is functioning as expected with a test recipe provided by the astrodata_Gemini package:

```
$ reduce --recipe test_one /path/to/gemini_data.fits
```

If all is well, users will see something like:

```
Resetting logger for application: reduce
Logging configured for application: reduce
                        --- reduce, v4890  ---
              Running under astrodata Version GP-X1
All submitted files appear valid
Starting Reduction on set #1 of 1

  Processing dataset(s):
        gemini_data.fits

================================================================================
RECIPE: test_one
================================================================================
 PRIMITIVE: showParameters
 -------------------------
 rtf = False
 suffix = '_scafaasled'
 otherTest = False
 logindent = 3
 logfile = 'reduce.log'
 reducecache = '.reducecache'
 storedcals = 'calibrations/storedcals'
 index = 1
 retrievedcals = 'calibrations/retrievedcals'
 cachedict = {'storedcals': 'calibrations/storedcals', 'retrievedcals':
              'calibrations/retrievedcals', 'calibrations': 'calibrations',
              'reducecache': '.reducecache'}
 loglevel = 'stdinfo'
 calurl_dict = {'CALMGR': 'http://fits/calmgr',
                'UPLOADPROCCAL': 'http://fits/upload_processed_cal',
                'QAMETRICURL': 'http://fits/qareport',
                'QAQUERYURL': 'http://fits/qaforgui',
                'LOCALCALMGR': 'http://localhost:%(httpport)d/calmgr/%(caltype)s'}
 logmode = 'standard'
 test = True
 writeInt = False
 calibrations = 'calibrations'
 .
Wrote gemini_data.fits in output directory


reduce completed successfully.
```

Users curious about the URLs in the example above, i.e. `http://fits/...`, see Sec. *Fits Storage* in Chapter 5, Discussion.

CHAPTER

THREE

INTERFACES

3.1 Introduction

The `reduce` application provides a command line interface and an API, both of which can configure and launch a Recipe System processing pipeline (a 'recipe') on the input dataset. Control of `reduce` and the Recipe System is provided by a variety of options and switches. Of course, all options and switches can be accessed and controlled through the API.

3.2 Command line interface

We begin with the command line help provided by `reduce --help`, followed by further description and discussion of certain non-trivial options that require detailed explanation.

```
usage: reduce [-h] [-v] [-d] [--context CONTEXT] [--logmode LOGMODE]
              [--logfile LOGFILE] [--loglevel LOGLEVEL]
              [-p USERPARAM [USERPARAM ...]] [-r RECIPENAME]
              [--user_cal USER_CAL] [--suffix SUFFIX]
              fitsfile [fitsfile ...]

_________________________________ Gemini Observatory _________________________________
_____________________ Recipe Processing Management System _____________________
____________________ recipeSystem2 Release alpha (new_hope) ____________________

positional arguments:
  fitsfile              fitsfile [fitsfile ...]

optional arguments:
  -h, --help            show this help message and exit
  -v, --version         show program's version number and exit
  -d , --displayflags   display all parsed option flags and exit.
  --context CONTEXT     Use <context> for recipe selection.
  --logmode LOGMODE     log mode: 'standard', 'console', 'quiet', 'debug','null'.
  --logfile LOGFILE     name of log (default is 'reduce.log')
  --loglevel LOGLEVEL   Set the verbose level for console logging.
  -p USERPARAM [USERPARAM ...], --param USERPARAM [USERPARAM ...]
                        Set a parameter from the command line.
  -r RECIPENAME, --recipe RECIPENAME
                        Specify a recipe by name.
  --user_cal USER_CAL   Specify user supplied calibrations.
  --suffix SUFFIX       Add 'suffix' to filenames at end of reduction.
```

The [options] are described in the following sections.

3.2.1 Informational switches

-h, –help show the help message and exit

-v, –version show program's version number and exit

-d, –displayflags Display all parsed option flags and exit.

When specified, this switch will present the user with a table of all parsed arguments and then exit without running. This allows the user to check that the configuration is as intended. The table provides a convenient view of all passed and default values. Unless a user has specified a recipe (-r, –recipe), 'recipename' indicates 'None' because at this point, the Recipe System has not yet been engaged and a default recipe not yet determined.

Eg.,:

```
$ reduce -d --logmode console fitsfile.fits

--------------------   switches, vars, vals  --------------------

Literals                  var 'dest'              Value
------------------------------------------------------------------
['-d', '--displayflags'] :: displayflags         :: True
['-p', '--param']        :: userparam            :: None
['--logmode']            :: logmode              :: ['console']
['-r', '--recipe']       :: recipename           :: None
['--logfile']            :: logfile              :: reduce.log
['--user_cal']           :: user_cal             :: None
['--context']            :: context              :: None
['--suffix']             :: suffix               :: None
['--loglevel']           :: loglevel             :: stdinfo
------------------------------------------------------------------

Input fits file(s):      fitsfile.fits
```

3.2.2 Configuration Switches, Options

–context <CONTEXT> Use <CONTEXT> for recipe selection and for primitives sensitive to context. Eg., `--context QA`. When not specified, the context defaults to 'QA'.

–logmode <LOGMODE> Set logging mode. One of

- standard
- console
- quiet
- debug
- null

where 'console' writes only to screen and 'quiet' writes only to the log file. Default is 'standard'.

–logfile <LOGFILE> Set the log file name. Default is 'reduce.log' in the current directory.

–loglevel <LOGLEVEL> Set the verbose level for console logging. One of

- critical
- error
- warning

- status
- stdinfo
- fullinfo
- debug

Default setting is 'stdinfo.'

–user_cal <USER_CAL [USER_CAL ...]> The option allows users to provide their own calibrations to `reduce`. Add a calibration to User Calibration Service. '–override_cal CAL_PATH' Eg.,

```
--user_cal wcal/gsTest_arc.fits
```

-p <USERPARAM [USERPARAM ...]>, –param <USERPARAM [USERPARAM ...]> Set a primitive parameter from the command line. The form '-p par=val' sets the parameter in the reduction context such that all primitives will 'see' it. The form

```
-p primitivename:par=val
```

sets the parameter such that it applies only when the primitive is 'primitivename'. Separate parameter-value pairs by whitespace: (eg. '-p par1=val1 par2=val2')

See Sec. *Overriding Primitive Parameters*, for more information on these values.

-r <RECIPENAME>, –recipe <RECIPENAME> Specify an explicit recipe to be used rather than internally determined by a dataset's <ASTROTYPE>. Default is None and later determined by the Recipe System based on the AstroDataType.

–suffix <SUFFIX> Add 'suffix' to output filenames at end of reduction.

3.2.3 Nominal Usage

The minimal call for reduce can be

```
$ reduce <dataset.fits>
```

While this minimal call is available at the Gemini Observatory (see Sec. *Fits Storage*), if a calibration service is unavailable to the user – likely true for most users – users should call `reduce` on a specified dataset by providing calibration files with the –user_cal option.

For example:

```
$ reduce --user_cal FOO_bias.fits <dataset.fits>
```

Such a command for complex processing of data is possible because AstroData and the Recipe System do all the necessary work in determining how the data are to be processed, which is critcially based upon the determination of the *tag set* that applies to that data.

Without any user-specified recipe (-r –recipe), the default recipe is `qaReduce`, which is defined for various AstroData tag sets and currently used during summit operations. Unless passed a explicit recipe (-r –recipename), the Recipe System uses the astrodata tag set and context to locate the appropriate recipe to run.

The recipe libraries for a GMOS_IMAGE, are defined under

```
GMOS.recipes.QA
```

and the recipe system will search available recipe libraries for a match. Naming of recipe library module(s) is arbitrary. If all defaults are picked up, this results in the `qaReduce` recipe function being selected and which specifies that the following primitives are called on the data

```
def qaReduce(p):
    p.prepare()
    p.addDQ()
    p.addVAR(read_noise=True)
    p.detectSources()
    p.measureIQ(display=True))
    p.measureBG()
    p.measureCCAndAstrometry()
    p.overscanCorrect()
    p.biasCorrect()
    p.ADUToElectrons()
    p.addVAR(poisson_noise=True)
    p.flatCorrect()
    p.mosaicDetectors()
    p.makeFringe()
    p.fringeCorrect()
    p.detectSources()
    p.measureIQ(display=True))
    p.measureBG()
    p.measureCCAndAstrometry()
    p.addToList(purpose=forStack)
```

The point here is not to overwhelm readers with a stack of primitive names, but to present both the default pipeline processing that the above simple `reduce` command invokes and to demonstrate how much the `reduce` interface abstracts away the complexity of the processing that is engaged with the simplicity of commands.

3.2.4 Overriding Primitive Parameters

In some cases, users may wish to change the functional behaviour of certain processing steps, i.e. change default behaviour of primitive functions.

Each primitive has a set of pre-defined parameters, which are used to control functional behaviour of the primitive. Each defined parameter has a "user override" token, which indicates that a particular parameter may be overridden by the user. Users can adjust parameter values from the reduce command line with the option,

-p, –param

If permitted by the "user override" token, parameters and values specified through the **-p, –param** option will *override* the defined parameter default value and may alter default behaviour of the primitive accessing this parameter. A user may pass several parameter-value pairs with this option.

Eg.:

```
$ reduce -p par1=val1 par2=val2 [par3=val3 ... ] <fitsfile1.fits>
```

User-specified parameter values can be focused on one primitive. For example, if a parameter applies to more than one primitive, for example, the parameter, `threshold`, the user can explicitly direct a new parameter value to a particular primitive. The 'detection threshold' has a defined default, but a user may alter this parameter default to change the source detection behaviour:

```
$ reduce -p detectSources:threshold=4.5 <fitsfile.fits>
```

3.2.5 The @file facility

The reduce command line interface supports what might be called an 'at-file' facility (users and readers familiar with IRAF will recognize this facility). This facility allows users to provide any and all command line options and flags to `reduce` via in a single acsii text file.

By passing an @file to `reduce` on the command line, users can encapsulate all the options and positional arguments they might wish to specify in a single @file. It is possible to use multiple @files and even to embed one or more @files in another. The parser opens all files sequentially and parses all arguments in the same manner as if they were specified on the command line. Essentially, an @file is some or all of the command line and parsed identically.

To illustrate the convenience provided by an '@file', let us begin with an example *reduce* command line that has a number of arguments:

```
$ reduce -p detectSources:threshold=4.5 tpar=100 -r recipe.ArgsTest --context SQ
  S20130616S0019.fits N20100311S0090.fits
```

Ungainly, to be sure. Here, two (2) *user parameters* are being specified with **-p**, a *recipe* with **-r**, and a *context* argument is specified to be **qa** . This can be wrapped in a plain text @file called *reduce_args.par*:

```
S20130616S0019.fits
N20100311S0090.fits
--param
tpar=100
detectSources:threshold=4.5
-r recipe.ArgsTests
--context sq
```

This then turns the previous reduce command line into something a little more *keyboard friendly*:

```
$ reduce @reduce_args.par
```

The order of these arguments is irrelevant. The above file could be thus written like:

```
-r recipe.ArgsTests
--param
tpar=100
detectSources:threshold=4.5
--context qa
S20130616S0019.fits
N20100311S0090.fits
```

Comments are accommodated, both as full line and in-line with the # character. White space is the only significant separator of arguments: spaces, tabs, newlines are all equivalent when argument parsing. This means the user can "arrange" their @file for clarity.

Here's a more readable version of the file from the previous example using comments and tabulation:

```
# reduce parameter file
# GDPSG

# Spec the recipe
-r
    recipe.ArgsTests  # test recipe

# primitive parameters here
--param
    tpar=100
    detectSources:threshold=4.5

--context
    qa                # QA context

S20130616S0019.fits
N20100311S0090.fits
```

All the above examples of `reduce_args.par` are equivalently parsed, which users may check by adding the **-d** flag:

```
$ reduce -d @redpars.par

--------------------   switches, vars, vals  --------------------

Literals                    var 'dest'          Value
------------------------------------------------------------------
['--invoked']               :: invoked          :: False
['-d', '--displayflags']    :: displayflags     :: True
['-p', '--param']           :: userparam        :: ['tpar=100', 'detectSources:threshold=4.5']
['--logmode']               :: logmode          :: standard
['-r', '--recipe']          :: recipename       :: ['recipe.ArgTests']
['--logfile']               :: logfile          :: reduce.log
['--user_cal']              :: user_cals        :: None
['--context']               :: context          :: ['QA']
['--calmgr']                :: cal_mgr          :: None
['--suffix']                :: suffix           :: None
['--loglevel']              :: loglevel         :: stdinfo
------------------------------------------------------------------

Input fits file(s):   S20130616S0019.fits
Input fits file(s):   N20100311S0090.fits
```

3.2.6 Recursive @file processing

As implemented, the @file facility will recursively handle, and process correctly, other @file specifications that appear in a passed @file or on the command line. For example, we may have another file containing a list of fits files, separating the command line flags from the positional arguments.

We have a plain text 'fitsfiles' containing the line:

```
test_data/S20130616S0019.fits
```

We can indicate that this file is to be consumed with the prefix character "@" as well. In this case, the 'reduce_args.par' file could thus appear:

```
# reduce test parameter file

@fitsfiles       # file with fits files

# primitive parameters.
--param
    detectSources:threshold=4.5
    tpar=99
    FOO=BAR

# Spec the recipe
-r recipe.ArgTests
```

The parser will open and read the @fitsfiles, consuming those lines in the same way as any other command line arguments. Indeed, such a file need not only contain fits files (positional arguments), but other arguments as well. This is recursive. That is, the @fitsfiles can contain other at-files", which can contain other "at-files", which can contain ..., etc. These will be processed serially.

As stipulated earlier, because the @file facility provides arguments equivalent to those that appear on the command line, employment of this facility means that a reduce command line could assume the form:

```
$ reduce @parfile @fitsfiles
```

or equally:

```
$ reduce @fitsfiles @parfile
```

where 'parfile' could contain the flags and user parameters, and 'fitsfiles' could contain a list of datasets.

Eg., fitsfiles comprises the one line:

```
test_data/N20100311S0090.fits
```

while parfile holds all other specifications:

```
# reduce test parameter file
# GDPSG

# primitive parameters.
--param
    detectSources:threshold=4.5
    tpar=99            # This is a test parameter
    FOO=BAR            # This is a test parameter

# Spec the recipe
-r recipe.ArgTests
```

The @file does not need to be located in the current directory. Normal, directory path syntax applies, for example:

```
reduce @../../mydefaultparams @fitsfile
```

3.2.7 Overriding @file values

The `reduce` application employs a customized command line parser such that the command line option

-p or **–param**

will accumulate a set of parameters *or* override a particular parameter. This may be seen when a parameter is specified in a user @file and then specified on the command line. For unitary value arguments, the command line value will *override* the @file value.

It is further specified that if one or more datasets (i.e. positional arguments) are passed on the command line, *all fits files appearing as positional arguments in the parameter file will be replaced by the command line arguments.*

Using the parfile above,

Eg. 1) Accumulate a new parameter:

```
$ reduce @parfile --param FOO=BARSOOM

parsed options:
---------------
FITS files:    ['S20130616S0019.fits', 'N20100311S0090.fits']
Parameters:    tpar=100, detectSources:threshold=4.5, FOO=BARSOOM
RECIPE:        recipe.ArgsTest
```

Eg. 2) Override a parameter in the @file:

```
$ reduce @parfile --param tpar=99

parsed options:
```

```
---------------
FITS files:     ['S20130616S0019.fits', 'N20100311S0090.fits']
Parameters:     tpar=99, detectSources:threshold=4.5
RECIPE:         recipe.ArgsTest
```

Eg. 3) Override the recipe:

```
$ reduce @parfile -r=recipe.FOO

parsed options:
---------------
FITS files:     ['S20130616S0019.fits', 'N20100311S0090.fits']
Parameters:     tpar=100, detectSources:threshold=4.5
RECIPE:         recipe.FOO
```

Eg. 4) Override a recipe and specify another fits file. The file names in the @file will be ignored:

```
$ reduce @parfile -r=recipe.FOO test_data/N20100311S0090_1.fits

parsed options:
---------------
FITS files:     ['test_data/N20100311S0090_1.fits']
Parameters:     tpar=100, detectSources:threshold=4.5
RECIPE:         recipe.FOO
```

3.3 Application Programming Interface (API)

Note: This section describes and discusses the programmatic interface available on the class Reduce. This section is for advanced users wishing to code using the `Reduce` class, rather than using `reduce` at the command line.

The `reduce` application is essentially a skeleton script providing the described command line interface. After parsing the command line, the script then passes the parsed arguments to its main() function, which in turn calls the Reduce() class constructor with "args". The Reduce class is scriptable by any user as the following discussion illustrates.

3.3.1 Class Reduce, logging, and the runr() method

The Reduce class is defined under the `gemini_python` code base in the `recipe_system.reduction` module, `coreReduce.py`.

The Reduce() class is importable and provides settable attributes and a callable that can be used programmatically. Callers need not supply an "args" parameter to the class initializer, i.e. __init__(). An instance of Reduce will have all the same arguments as in a command line scenario, available as attributes on the instance. Once an instance of Reduce() is instantiated and instance attributes set as needed, there is one (1) method to call, **runr**(). This is the only public method on the class.

Eg.,

```
>>> from recipe_system.reduction.coreReduce import Reduce
>>> reduce = Reduce()
>>> reduce.files
[]
>>> reduce.files.append('S20130616S0019.fits')
>>> reduce.files
['S20130616S0019.fits']
```

Or callers may simply set the `files` attribute to be an existing list of files

```
>>> fits_list = ['FOO.fits', 'BAR.fits']
>>> reduce.files = fits_list
```

On the command line, users may specify a recipe with the `-r` [`--recipe`] flag. Programmatically, users directly set the recipe:

```
>>> reduce.recipename = 'recipe.MyRecipe'
```

All other properties and attributes on the API may be set in standard pythonic ways. See Appendix *Class Reduce: Settable properties and attributes* for further discussion and more examples.

Using the logger

Note: When using an instance of Reduce() directly, callers must configure their own logger. Reduce() does not configure logutils prior to using a logger as returned by logutils.get_logger(). The following discussion demonstrates how this is easily done. It is *highly recommended* that callers configure the logger.

It is recommended that callers of Reduce use a logger supplied by the astrodata module `logutils`. This module employs the python logger module, but with recipe system specific features and embellishments. The recipe system expects to have access to a logutils logger object, which callers should provide prior to calling the `runr()` method.

To use `logutils`, import, configure, and get it:

```
from gempy.utils import logutils
logutils.config()
log = logutils.get_logger(__name__)
```

where `__name__` is usually the calling module's __name__ property, but can be any string value. Once configured and instantiated, the `log` object is ready to use. See section *Configuration Switches, Options* for logging levels described on the `--loglevel` option.

Once an instance of Reduce has been made, callers may (should) configure the logutils facility with attributes available on the instance. Instances of `Reduce()` provide the following logger parameters as attributes on the instance with appropriate default values:

- logfile
- loglevel
- logmode
- logindent

The `reduce` command line provides access to the first three of these attributes, as described in Sec. *Configuration Switches, Options*, but `logindent`, which controls the indention levels of logging output, is accessible only through the public interface on an instance of `Reduce()`. It is not anticipated that users will need, or even want, to change the value of `logindent`, but it is possible.

An instance of `Reduce()` provides the following attributes that may be passed to the `logutils.config()`. The default values provided for these logging configuration parameters may be examined through direct inspection:

```
>>> reduce = Reduce()
>>> reduce.logfile
'reduce.log'
>>> reduce.logmode
'standard'
>>> reduce.loglevel
'stdinfo'
```

```
>>> reduce.logindent
3
```

Users may adjust these values and then pass them to the `logutils.config()` function, or pass other values directly to `config()`. This is precisely what `reduce` does when it configures logutils. See Sec. *Configuration Switches, Options* and Appendix *Class Reduce: Settable properties and attributes* for allowable and default values of these and other options.

```
>>> from gempy.utils import logutils
>>> logutils.config(file_name=reduce.logfile, mode=reduce.logmode,
                    console_lvl=reduce.loglevel)
```

Note: logutils.config() may be called mutliply, should callers, for example, want to change logfile names for different calls on runr().

Call the runr() method

Once a user is satisfied that all attributes are set to the desired values, and the logger is configured, the runr() method on the "reduce" instance may then be called. The following brings the examples above into one "end-to-end" use of Reduce and logutils:

```
>>> from recipe_system.reduction.coreReduce import Reduce
>>> from gempy.utils import logutils
>>> reduce = Reduce()
>>> reduce.files.append('S20130616S0019.fits')
>>> reduce.recipename = 'recipe.MyRecipe'
>>> reduce.logfile = 'my_reduce_run.log'
>>> logutils.config(file_name=reduce.logfile, mode=reduce.logmode,
                    console_lvl=reduce.loglevel)
>>> reduce.runr()
All submitted files appear valid
Starting Reduction on set #1 of 1
Processing dataset(s):
S20130616S0019.fits
...
```

Processing will then proceed in the usual manner. Astute readers will note that callers need not create more than one Reduce instance in order to call runr() with a different dataset or options.

Eg.,:

```
>>> from recipe_system.reduction.coreReduce import Reduce
>>> from gempy.utils import logutils
>>> reduce = Reduce()
>>> reduce.files.append('S20130616S0019.fits')
>>> reduce.recipename = 'recipe.MyRecipe'
>>> reduce.logfile = 'my_reduce_run.log'
>>> logutils.config(file_name=reduce.logfile, mode=reduce.logmode,
                     console_lvl=reduce.loglevel)
>>> reduce.runr()
  ...
reduce completed successfully.

>>> reduce.recipename = 'recipe.NewRecipe'
>>> reduce.files = ['newfile.fits']
>>> reduce.userparam = ['clobber=True']
>>> runr()
```

Once an attribute is set on an instance, such as above with `userparam`, it is always set on the instance. If, on another call of runr() the caller does not wish to have `clobber=True`, simply reset the property:

```
>>> reduce.userparam = []
>>> runr()
```

Readers may wish to examine the examples in Appendix *Class Reduce: Settable properties and attributes*

CHAPTER

FOUR

SUPPLEMENTAL TOOLS

The astrodata package provides a number of command line driven tools, which users may find helpful in executing reduce on their data.

With the installation and configuration of `astrodata` and `reduce` comes some supplemental tools to help users discover information, not only about their own data, but about the Recipe System, such as available recipes, primitives, and defined AstroDataTypes.

If the user environment has been configured correctly these applications will work directly.

4.1 typewalk

`typewalk` examines files in a directory or directory tree and reports the types and status values through the AstroDataType classification scheme. Running `typewalk` on a directory containing some Gemini datasets will demonstrate what users can expect to see. If a user has downloaded gemini_python package with the 'test_data', the user can move to this directory and run `typewalk` on that extensive set of Gemini datasets.

By default, `typewalk` will recurse all subdirectories under the current directory. Users may specify an explicit directory with the **-d** or **–dir** option; the behavior remains recursive.

`typewalk` provides the following options [**-h, –help**]:

```
-h, --help            show this help message and exit
-b BATCHNUM, --batch BATCHNUM
                      In shallow walk mode, number of files to process at a
                      time in the current directory. Controls behavior in
                      large data directories. Default = 100.
-d TWDIR, --dir TWDIR
                      Walk this directory and report types. default is cwd.
-f FILEMASK, --filemask FILEMASK
                      Show files matching regex <FILEMASK>. Default is all
                      .fits and .FITS files.
-n, --norecurse       Do not recurse subdirectories.
--or                  Use OR logic on 'types' criteria. If not specified,
                      matching logic is AND (See --types). Eg., --or --types
                      SOUTH GMOS IMAGE will report datasets that are one of
                      SOUTH *OR* GMOS *OR* IMAGE.
-o OUTFILE, --out OUTFILE
                      Write reported files to this file. Effective only with
                      --tags option.
--tags TAGS [TAGS ...]
                      Find datasets that match only these tag criteria. Eg.,
                      --tags SOUTH GMOS IMAGE will report datasets that are
                      all tagged SOUTH *and* GMOS *and* IMAGE.
```

```
--xtags XTAGS [XTAGS ...]
                      Exclude <xtags> from reporting.
```

Files are selected and reported through a regular expression mask which, by default, finds all ".fits" and ".FITS" files. Users can change this mask with the **-f, –filemask** option.

As the **–types** option indicates, `typewalk` can find and report data that match specific type criteria. For example, a user might want to find all GMOS image flats under a certain directory. `typewalk` will locate and report all datasets that would match the AstroDataType, GMOS_IMAGE_FLAT.

A user may request that a file be written containing all datasets matching AstroDataType qualifiers passed by the **–types** option. An output file is specified through the **-o, –out** option. Output files are formatted so they may be passed *directly to the reduce command line* via that applications 'at-file' (@file) facility. See *The @file facility* or the reduce help for more on 'at-files'.

Users may select type matching logic with the **–or** switch. By default, qualifying logic is AND, i.e. the logic specifies that *all* types must be present (x AND y); **–or** specifies that ANY types, enumerated with **–types**, may be present (x OR y). **–or** is only effective when the **–types** option is specified with more than one type.

For example, find all GMOS images from Cerro Pachon in the top level directory and write out the matching files, then run reduce on them (**-n** is 'norecurse'):

```
$ typewalk -n --tags SOUTH GMOS IMAGE --out gmos_images_south
$ reduce @gmos_images_south
```

Find all F2 SPECT datasets in a directory tree:

```
$ typewalk --tags SPECT F2
```

This will also report match results to stdout, colourized if requested (**-c**).

Users may find the **–xtypes** flag useful, as it provides a facility for filtering results further by allowing certain types to be excluded from the report.

For example, find GMOS_IMAGE types, but exclude ACQUISITION images from reporting:

```
$ typewalk --tags GMOS IMAGE --xtags ACQUISITION

directory: ../test_data/output
   S20131010S0105.fits ............... (GEMINI) (SOUTH) (GMOS) (IMAGE) (RAW)
   (SIDEREAL) (UNPREPARED)

   S20131010S0105_forFringe.fits ..... (GEMINI) (SOUTH) (GMOS)
   (IMAGE) (NEEDSFLUXCAL) (OVERSCAN_SUBTRACTED) (OVERSCAN_TRIMMED)
   (PREPARED) (PROCESSED_SCIENCE) (SIDEREAL)

   S20131010S0105_forStack.fits ...... (GEMINI) (SOUTH) (GMOS) (IMAGE)
   (NEEDSFLUXCAL) (OVERSCAN_SUBTRACTED) (OVERSCAN_TRIMMED)
   (PREPARED) (SIDEREAL)
```

Exclude GMOS ACQUISITION images and GMOS IMAGE datasets that have been 'prepared':

```
$ typewalk --tags GMOS IMAGE --xtags ACQUISITION PREPARED

directory: ../test_data/output
   S20131010S0105.fits ............... (GEMINI) (SOUTH) (GMOS) (IMAGE) (RAW)
   (SIDEREAL) (UNPREPARED)
```

With **–tags** and **–xtags**, users may really tune their searches for very specific datasets.

CHAPTER

FIVE

DISCUSSION

5.1 Fits Storage

The URLs that appear in `test_one` recipe example (Sec. *Test the installation*), reference web services available within the Gemini Observatory's operational environment. They will *not* be available directly to users running `reduce` outside of the Gemini Observatory environment.

In the context of `reduce` and the Astrodata Recipe System, FitsStorage provides a calibration management and association feature. Essentially, given a science frame (or any frame that requires calibration) and a calibration type requested, FitsStorage is able to automatically choose the best available calibration of the required type to apply to the science frame. The Recipe System uses a machine-oriented calibration manager interface in order to select calibration frames to apply as part of pipeline processing.

Though this service is not currently available to general gemini_python users, plans to provide this as a local calibration service are in place and expected for *Future Enhancements*.

5.2 Future Enhancements

5.2.1 Intelligence

One enhancement long imagined is what has been generally termed 'intelligence'. That is, an ability for either `reduce` or some utility to automatically do AstroDataType classification of a set of data, group them appropriately, and then pass these grouped data to the Recipe System.

As things stand now, it is up to the user to pass commonly typed data to `reduce`. As shown in the previous section, *typewalk*, `typewalk` can help a user perform this task and create a 'ready-to-run' @file that can be passed directly to `reduce`. Properly implemented 'intelligence' will *not* require the user to determine the AstroDataTypes of datasets.

5.2.2 Local Calibration Service

The Fits Storage service will be delivered as part of a future release and will provide the calibration management and association features of *Fits Storage*: for use with the public release of the *gemini_python* data reduction package. This feature will provide automatic calibration selection for both pipeline (recipe) operations and in an interactive processing environment.

CHAPTER

SIX

6. ACKNOWLEDGMENTS

The Gemini Observatory is operated by the Association of Universities for Research in Astronomy (AURA), Inc., under a cooperative agreement with the NSF on behalf of the Gemini partnership: the National Science Foundation (United States), the Science and Technology Facilities Council (United Kingdom), the National Research Council (Canada), CONICYT (Chile), the Australian Research Council (Australia), Ministerio da Ciencia e Tecnologia (Brazil), and Ministerio de Ciencia, Tecnologia e Innovacion Productiva (Argentina).

APPENDIX

A

REDUCE DEMO

Original demo author: Kathleen Labrie, October 2014

A.1 Setting up

First install Ureka, which can be obtained at http://ssb.stsci.edu/ureka/.

The second step is to install `gemini_python` as described in *Section 2 - Installation*. Please do make sure that the command *reduce* is in your `PATH` and that `PYTHONPATH` includes the location where the modules `astrodata`, the `recipe_system`, and `gempy` are installed.

The demo data is distributed separately. You can find the demo data package `gemini_python_datapkg-X1.tar.gz` on the Gemini website where you found the gemini_python package. Unpack the data package somewhere convenient:

```
tar xvzf gemini_python_datapkg-X1.tar.gz
```

In there, you will find a subdirectory named `data_for_reduce_demo`. Those are the data we will use here. You will also find an empty directory called `playground`. This is your playground. The instructions in this demo assume that you are running the `reduce` command from that directory. There is no requirements to run `reduce` from that directory, but if you want to follow the demo to the letter, this is where you should be for all the paths to work.

A.2 Introduction to the Demo

In this demo, we will reduce a simple dither-on-source GMOS imaging sequence. We will first process the raw biases, and then the raw twilight flats. We will then use those processed files to process and stack the science observation.

Instead of the default Quality Assessment (QA) recipe that is used at the Gemini summits, we will use another recipe that will focus on the reduction rather than on the multiple measurements of the QA metrics used at night. QA metrics, here the image quality (IQ), will only be measured at the end of the reduction rather than throughout the reduction. Another difference between the standard QA recipe and the demo recipe, is that the demo recipe does stack the data, while the stacking is turned off in the QA context.

The demo recipe is essentially a Quick Look recipe. It is NOT valid for Science Quality. Remember that what you are using is a QA pipeline, not a Science pipeline.

A.3 The Recipes

To process the biases and the flats we will be using the standard recipes. The system will be able to pick those automatically when it recognizes the input data as GMOS biases and GMOS twilight flats.

For the science data, we will override the recipe selection to use the Demo recipe. If we were not to override the recipe selection, the system would automatically select the QA recipe. The Demo recipe is more representative of a standard Quick-Look reduction with stacking, hence probably more interesting to the reader.

The standard recipe to process GMOS biases is named `makeProcessedBias` and contains the instruction set:

```
# This recipe performs the standardization and corrections needed to convert
# the raw input bias images into a single stacked bias image. This output
# processed bias is stored on disk using storeProcessedBias and has a name
# equal to the name of the first input bias image with "_bias.fits" appended.

p.prepare()
p.addDQ()
p.addVAR(read_noise=True)
p.overscanCorrect()
p.addToList(purpose="forStack")
p.getList(purpose="forStack")
p.stackFrames()
p.storeProcessedBias()
```

The standard recipe to process GMOS twilight flats is named `makeProcessedFlat` and contains the instruction set:

```
# This recipe performs the standardization and corrections needed to convert
# the raw input flat images into a single stacked and normalized flat image.
# This output processed flat is stored on disk using storeProcessedFlat and
# has a name equal to the name of the first input flat image with "_flat.fits"
# appended.

p.prepare()
p.addDQ()
p.addVAR(read_noise=True)
p.display()
p.overscanCorrect()
p.biasCorrect()
p.ADUToElectrons()
p.addVAR(poisson_noise=True)
p.addToList(purpose="forStack")
p.getList(purpose="forStack")
p.stackFlats()
p.normalizeFlat()
p.storeProcessedFlat()
```

The Demo recipe is named `reduceDemo` and contains the instruction set:

```
# reduceDemo

p.prepare()
p.addDQ()
p.addVAR(read_noise=True)
p.overscanCorrect()
p.biasCorrect()
p.ADUToElectrons()
p.addVAR(poisson_noise=True)
p.flatCorrect()
p.makeFringe()
p.fringeCorrect()
p.mosaicDetectors()
p.detectSources()
p.addToList(purpose=forStack)
```

```
p.getList(purpose=forStack)
p.alignAndStack()
p.detectSources()
p.measureIQ()
```

For the curious, the standard bias and flat recipes are found in `???` and the demo recipe is in `???demos/`. You do not really need that information as the system will find them on its own.

A.4 The Demo

The images will be displayed at times. Therefore, start ds9:

```
ds9 &
```

A.4.1 The Processed Bias

The first step is to create the processed bias. We are using the standard recipe. The system will recognize the inputs as GMOS biases and call the appropriate recipe automatically.

The biases were taken on different dates around the time of the science observations. For convenience, we will use a file with the list of datasets as input instead of listing all the input datasets individually. We will use a tool named `typewalk` to painlessly create the list.

```
cd <your_path>/gemini_python_datapkg-X1/playground

typewalk --types GMOS BIAS --dir ../data_for_reduce_demo -o bias.list

reduce @bias.list
```

This creates the processed bias, `N20120202S0955_bias.fits`. The output suffix `_bias` is the indicator that this is a processed bias. All processed calibrations are also stored in `./calibrations/storedcals/` for safe keeping.

If you wish to see what the processed bias looks like:

```
reduce N20120202S0955_bias.fits -r display
```

Note: This will issue an error about the file already existing. Ignore it. The explanation of what is going on is beyond the scope of this demo. We will fix this, eventually. Remember that this is a release of software meant for internal use; there are still plenty of issues to be resolved.

A.4.2 The Processed Flat

Next we create a processed flat. We will use the processed bias we have just created. The system will recognize the inputs as GMOS twilight flats and call the appropriate recipe automatically.

The "public" RecipeSystem does not yet have a Local Calibration Server. Therefore, we will need to specify the processed bias we want to use on the *reduce* command line. For information only, internally the QA pipeline at the summit uses a central calibration server and the most appropriate processed calibrations available are selected and retrieved automatically. We hope to be able to offer a "local", end-user version of this system in the future. For now, calibrations must be specified on the command line.

For the flats, we do not really need a list, we can use wild cards:

```
reduce ../data_for_reduce_demo/N20120123*.fits \
   --user_cal N20120202S0955_bias.fits -p clobber=True;
```

This creates the processed flat, `N20120123S0123_flat.fits`. The output suffix `_flat` is the indictor that this is a processed flat. The processed flat is also stored in `./calibrations/storedcals/` for safe keeping.

The `clobber` parameter is set to True to allow the system to overwrite the final output. By default, the system refuses to overwrite an output file.

If you wish to see what the processed flat looks like:

```
reduce N20120123S0123_flat.fits -r display
```

A.4.3 The Science Frames

We now have all the pieces required to reduce the science frames. This time, instead of using the standard QA recipe, we will use the Demo recipe. Again, we will specify the processed calibrations, bias and flat, we wish to use.

```
reduce ../data_for_reduce_demo/N20120203S028?.fits \
   --user_cal N20120202S0955_bias.fits N20120123S0123_flat.fits \
   -r reduceDemo -p clobber=True
```

The demo data was obtained with the z' filter, therefore the images contain fringing. The `makeFringe` and `fringeCorrect` primitives are filter-aware, they will do something only when the data is from a filter that produces fringing, like the z' filter. The processed fringe that is created is stored with the other processed calibrations in `./calibrations/storedcals/` and it is named `N20120203S0281_fringe.fits`. The `_fringe` suffix indicates a processed fringe.

The last primitive in the recipe is `measureIQ` which is one of the QA metrics primitives used at night by the QA pipeline. The primitive selects stars in the field and measures the average seeing and ellipticity. The image it runs on is displayed and the selected stars are circled for visual inspections.

The fully processed stacked science image is `N20120203S0281_iqMeasured.fits`. By default, the suffix of the final image is set by the last primitive run on the data, in this case `measureIQ`.

This default naming can be confusing. If you wish to set the suffix of the final image yourself, use `--suffix _myfinalsuffix`.

A.4.4 Clean up

It is good practice to reset the RecipeSystem state when you are done:

```
superclean --safe
```

Your files will stay there, only some hidden RecipeSystem directories and files will be deleted.

A.5 Limitations

The X1 version of the RecipeSystem has not been vetted for Science Quality. Use ONLY for quick look purposes.

The RecipeSystem currently does not handle memory usage in a very smart way. The number of files one can pass on to `reduce` is directly limited by the memory of the user's computer. This demo ran successfully on a Mac laptop with 4 GB of memory.

APPENDIX

B

CLASS REDUCE: SETTABLE PROPERTIES AND ATTRIBUTES

The public interface on an instance of the Reduce() class provides a number of properties and attributes that allow the user to set and reset options as they might through the reduce command line interface. The following table is an enumerated set of those attributes.

An instance of Reduce() provides the following attributes. (Note: defaults are not necessarily indicative of the actual type that is expected on the instance. Use the type specified in the type column.):

```
Attribute               Python type         Default
-------------------------------------------------------
displayflags            <type 'bool'>        False
files                   <type 'list'>        []
context                 <type 'str'>         None
logfile                 <type 'str'>         'reduce.log'
loglevel                <type 'str'>         'stdinfo'
logmode                 <type 'str'>         'standard'
recipename              <type 'str'>         None
suffix                  <type 'str'>         None
user_cal                <type 'str'>         None
userparam               <type 'list'>        None
```

B.1 Examples

Setting attributes on a Reduce object:

```
>>> reduce = ReduceNH()
>>> reduce.logfile = "my_reduction.log"
>>> reduce.recipe = "recipe.my_recipe"
>>> reduce.files = ['UVW.fits', 'XYZ.fits']
```

Or in other pythonic ways:

```
>>> file_list = ['FOO.fits', 'BAR.fits']
>>> reduce.files.extend(file_list)
>>> reduce.files
['UVW.fits', 'XYZ.fits', 'FOO.fits', 'BAR.fits']
```

Users wishing to pass primtive parameters to the recipe_system need only set the one (1) attribute, `userparam`, on the Reduce instance:

```
>>> reduce.userparam = ['clobber=True']
```

This is the API equivalent to the command line option:

```
$ reduce -p clobber=True [...]
```

For muliple primitive parameters, the 'userparam' attribute is a list of 'par=val' strings, as in:

```
>>> reduce.userparam = [ 'par1=val1', 'par2=val2', ... ]
```

B.2 Example function

The following function shows a potential usage of class Reduce. When conditions are met, the function `reduce_conditions_met()` is called passing several lists of files, `procfiles` (a list of lists of fits files). Here, each list of `procfiles` is then passed to the internal `launch_reduce()` function.

```
from gempy.utils import logutils
from recipe_systenm.reduction.coreReduce import Reduce

def reduce_conditions_are_met(procfiles, control_options={}):
    reduce_object = Reduce()
    reduce_object.logfile = 'my_reduce.log'
    # write logfile only, no stdout.
    reduce_object.logmode = 'quiet'
    reduce_object.userparam = ['clobber=True']

    logutils.config(file_name=reduce_object.logfile,
                    mode=reduce_object.logmode,
                    console_lvl=reduce_object.loglevel)

    def launch_reduce(datasets, recipe=None, upload=False):
        reduce_object.files = datasets
        if recipe:
            reduce_object.recipename = recipe
        if upload:
            reduce_object.context = 'qa, upload'
        else:
            reduce_object.context = 'qa'
        reduce_object.runr()
        return

    for files in procfiles:
        # Use a different recipe if FOO.fits is present
        if "FOO.fits" in files:
            launch_reduce(sorted(files), recipe="recipe.FOO")
        else:
            launch_reduce(sorted(files), upload=control_options.get('upload'))

    return

procfiles = [ ['FOO.fits', 'BAR.fits'],
              ['UVW.fits', 'XYZ.fits']
            ]
if conditions_are_met:
    reduce_conditions_are_met(procfiles)
```

Readers will see here that calling `reduce_conditions_are_met()` without the `control_options` parameter will result in the `running_contexts` attribute being set to `'qa'`.