DRAGONS - Recipe System User Manual

Release 3.0.2

Kenneth Anderson, Kathleen Labrie, Bruno Quint

July 2022

Contents

Document ID

PIPE-USER-109_RSUserManual

CHAPTER 1

Introduction

1.1 Overview

The DRAGONS Recipe System is Gemini Observatory's data processing automation platform. The Recipe System is designed to accommodate both stepwise, interactive data processing, and automated data reduction pipelines.

The Recipe System inspect the inputs and automatically associates the recipes and primitives most appropriate for those inputs. A primitive is a step in a reduction, for example `biasCorrect`. A recipe is a sequence of primitives. For the Gemini instruments, the collections of primitives and recipes are found in the `geminidr` package. It is possible to specify a different data reduction package.

The Recipe System relies on the Astrodata facility (`astrodata` package) to identify the input data and match them to the recipes and primitives. The Astrodata *tags* are the keys to the mapping. For the Gemini instruments, the Astrodata configurations are found in the `gemini_instruments` package. Again, it is possible to specify a different Astrodata configuration package.

The `reduce` command and programmatic access to the `Reduce` class are the principle ways DRAGONS users can employ the Recipe System to process and reduce their data. This document discusses a variety of examples of the `reduce` command line and the programmatic interface on the `Reduce` class.

The `reduce` command, and its programmatic interface, support options that allow users to select and "tune" input parameters data processing steps. Without any command line options or adjustment of the `Reduce` class option attributes, the reduction uses default recipes and default input parameters to the primitives. In the `geminidr` package, which support the Gemini instruments, the default recipes and primitive parametres have been optimized to give good results in most cases.

A typical `reduce` command can look deceptively simple. Without knowing the content of the data file, you can simply run `reduce` on the data and the Recipe System automatically selects the best recipe and primtives based upon the data classifications. For example, a call like this one can be all that is needed:

```
$ reduce S20161025S0111.fits
                        --- reduce, v2.0 (beta) ---
All submitted files appear valid
================================================================================
RECIPE: reduce
```

(continues on next page)

(continued from previous page)

```
================================================================================
 PRIMITIVE: prepare
 ------------------
 ...
 ...
```

1.2 Further Information

Details and information on developing for the Recipe System, and about the `astrodata` package are available in companion manuals. We invite the reader interested in those topics to refer to the topical documentation.

-
-
-

CHAPTER 2

Installation

The Recipe System is distributed as part of DRAGONS. DRAGONS is available as a conda package. The installation instructions below will install all the necessary dependencies.

The use of the `bash` shell is required by Anaconda.

2.1 Install Anaconda

If you already have Anaconda installed, you can skip this step and go to the *Install DRAGONS* section below. If not, then your first step is to get and install Anaconda. You can download it at:

Choose the version of Python that suits your other Python needs. DRAGONS is compatible with Python 3.7. We recommend that you install the standard Python 3 version of Anaconda, the specific Python version can be adjusted later.

If you have downloaded the graphical installer, follow the graphical installer instructions. Install in your home directory. It should be the default.

If you have downloaded the command-line installer, type the following in a terminal, replacing the `.sh` file name to the name of the file you have downloaded. The `/bin/bash -l` line is not needed if you are already using bash. The command-line installer allows for more customization of the installation.

```
$ /bin/bash -l
$ chmod a+x Anaconda3-2019.03-MacOSX-x86_64.sh
$ ./Anaconda3-2019.03-MacOSX-x86_64.sh
```

(`$` indicates the terminal prompt.)

Note: To prevent the Anaconda "base" environment from loading automatically, do:

```
$ conda config --set auto_activate_base false
```

2.2 Install DRAGONS

Anaconda requires the use of the bash shell. `tcsh` or `csh` will not work. If you are using (t)csh, your first step is:

```
$ /bin/bash -l
```

Make sure that `~/anaconda3/bin/activate` is in your `PATH` by doing:

```
$ which activate
```

The Anaconda installer should have added conda configurations to the `~/.bash_profile` for you. If `activate` is not found, try:

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

If `activate` is still not found, you might have to add `export PATH=~/anaconda3/bin:$PATH` to your `~/.bash_profile` using your favorite text editor, and run the `source` command above again.

Note: Sometimes the Anaconda installer will install the software in `~/anaconda3` instead of simply `~/anaconda`. Just check in your home directory which one of the tow possibilities was used.

The code Anaconda adds to the .bash_profile will automatically activate anaconda. To activate or deactivate Anaconda manually:

```
$ conda activate
$ conda deactivate
```

Now that Anaconda works, we add the needed astronomy software. Add the Astroconda channel and the Gemini channel. Those channels host the conda astronomy packages.

```
$ conda config --add channels http://ssb.stsci.edu/astroconda
$ conda config --add channels http://astroconda.gemini.edu/public
```

The next step is to create a virtual environment and install the DRAGONS software and its dependencies in it. The name of the environment can be anything you like. Here we use "dragons" as the name and we install Python 3.7.

```
$ conda create -n dragons python=3.7 dragons

Or, to include things like ds9

$ conda create -n dragons python=3.7 dragons stsci
```

Most users will probably want to install the extra astronomy tools that come with the `stsci` conda package.

To use this environment, activate it:

```
$ conda activate dragons
```

You will need to activate the environment whenever you start a new shell. If you are planning to use it all the time, you might want to add the command to your `.bash_profile`, after the "conda init" block.

Note: As a side note, if you are going to use PyRAF regularly, for example to reduce Gemini data not yet supported in DRAGONS, you should be installing Python 2.7 **as well** in a different environment, along with the `gemini`, `iraf-all` and `pyraf-all` conda packages. Do not use PyRAF from the Python 3 environment; PyRAF is very slow under Python 3.

$ conda create -n geminiconda python=2.7 iraf-all pyraf-all stsci gemini

DRAGONS and the Recipe System do not need IRAF, PyRAF. Only DRAGONS v2 is compatible with Python 2.7. See the Gemini website for information on how to configure IRAF ()

2.3 Configure DRAGONS

DRAGONS requires a configuration file located in `~/.geminidr/`:

```
$ cd ~
$ mkdir .geminidr
$ cd .geminidr
$ touch rsys.cfg
```

Open `rsys.cfg` with your favority editor and add these lines:

```
[calibs]
standalone = True
database_dir = ~/.geminidr/
```

Then configure `ds9` buffer configurations:

```
$ cd ~
$ cp $CONDA_PREFIX/lib/python3.7/site-packages/gempy/numdisplay/imtoolrc ~/.imtoolrc
$ vi .bash_profile   # or use your favorite editor

  Add this line to the .bash_profile:
    export IMTOOLRC=~/.imtoolrc
```

2.4 Test the installation

Start up the Python interpreter and import `astrodata` and the `gemini_instruments` packages:

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

If the imports are successful, i.e. no errors show up, exit Python (Ctrl-D).

Now test that `reduce` runs. There may be some delay as package modules are compiled and loaded:

```
$ reduce --help
```

This will print the reduce help to the screen.

If you have Gemini FITS files available, you can test that the Recipe System is functioning as expected as follow (replace the file name with the name of your file):

```
$ reduce N20180106S0700.fits -r prepare
```

If all is well, you will see something like:

```
                    --- reduce, v3.0.0 ---
All submitted files appear valid
Found 'prepare' as a primitive.
================================================================================
RECIPE: prepare
================================================================================
PRIMITIVE: prepare
------------------
  PRIMITIVE: validateData
  -----------------------
  .
  PRIMITIVE: standardizeStructure
  -------------------------------
  .
  PRIMITIVE: standardizeHeaders
  -----------------------------
     PRIMITIVE: standardizeObservatoryHeaders
     ----------------------------------------
     Updating keywords that are common to all Gemini data
     .
     PRIMITIVE: standardizeInstrumentHeaders
     ---------------------------------------
     Updating keywords that are specific to NIRI
     .
  .
.
Wrote N20180106S0700_prepared.fits in output directory

reduce completed successfully.
```

CHAPTER 3

Definitions

When a reduction is launched with `reduce` (command line) or `Reduce` (Python class), the Recipe System will identify the nature of the inputs using the AstroData tags, and then start searching for the most appropriate, or the requested, recipe and primitives.

The Recipe System will search the active data reduction package (`geminidr` or as specified by the `--drpkg` option) for recipe libraries and primitive sets matching the inputs. The recipe library search is limited in scope by the mode option.

Once everything has been found, the default or specified recipe from the selected recipe library is given the primitive set as input. The recipe is run and the sequence of primitive calls is executed.

Below, we discuss each of the terms in bold italics from the execution summary above: "AstroData tags", "mode", "recipe", "recipe library", "primitive", "primitive set".

3.1 AstroData Tags

The `AstroData Tags` are data identification tags. When a file is opened with `AstroData`, the software loads the *AstroData configuration files* and attempts to identify the data.

The tags associated with the dataset are compared to tags included in recipes and in primitive classes. The best match wins the selection process.

For Gemini instruments, the AstroData configurations are found in the `gemini_instruments` package. This is set as the default. Which configuration package to use can be configured on the `reduce` command line or in the `Reduce` class.

More information on AstroData tags can be found in the .

3.2 Mode

The `mode` defines the type of reduction one wants to perform: science quality ("sq"), quick look reduction ("ql"), or quality assessment ("qa"). Each `mode` defines its own set of recipe libraries. The mode is switched through command

line flags or the `Reduce` class `mode` attribute.

If not specified, the default is science quality, "sq". Currently, only science quality, quick look, and quality assessment are supported. Users cannot select other modes.

Recipe libraries of the same name but assigned different mode are often very different from each other since the products are expected to be different.

The quality assessment mode, "qa", is used mostly at the Observatory, at night to measure sky condition metrics and provide a visual assessment of the data. It does not require calibrations since we might not have all the calibrations needed at the time that the data was obtained.

The quick look mode, "ql", is intended for quick, close to but not necessarily science quality reduction. The objective as the name entails being to do a quick and automatic reduction for quick scientific and technical evaluation of the data. This mode does not require calibrations either, but both QA and QL modes can use calibrations if they are found.

The science quality mode, "sq", the default mode, is to be used in most cases. The recipes in "sq" mode contain all the steps required to fully reduce data without cutting corners. Some steps can be lengthy, some steps might offer an optional interactive interface for optimization. This mode requires all the calibrations and will return an error in case some of the is not found.

It is important to notice that a calibration processed with a mode cannot be used in another mode. So make sure you are reducing all your data using the same mode.

3.3 Recipe

A recipe is a sequence of data processing instructions. Technically, it is a Python function that calls a sequence of *primitives*, each primitive nominally designed to do one specific transformation or service request.

Below is what a recipe can look like. This recipe performs the standardization and corrections needed to convert the raw input science images into a stacked image. The argument, `p`, to the `reduce` recipe is the primitive set; the recipe can call any primitives from that set.

```
def reduce(p):
    p.prepare()
    p.addDQ()
    p.addVAR(read_noise=True)
    p.overscanCorrect()
    p.biasCorrect()
    p.ADUToElectrons()
    p.addVAR(poisson_noise=True)
    p.flatCorrect()
    p.mosaicDetectors()
    p.makeFringe()
    p.fringeCorrect()
    p.alignAndStack()
    p.writeOutputs()
    return
```

The guiding principle when building a recipe is to keep it human readable and scientifically oriented.

3.4 Recipe Library

A recipe library is a collection of recipes that applies to a specific type of data. The AstroData tags are used to match a recipe library to a dataset. A recipe library is implemented as Python module. There can be many recipes but only one is set as the default. It is however possible for the user to override the default and call any recipe within the library.

3.5 Primitive

A primitive is a data reduction step involving a transformation of the data or providing a service. By convention, the primitives are named to convey the scientific meaning of the transformation. For example `biasCorrect` will remove the bias signal from the input data.

A primitive is always a member of a primitive set. It is the primitive set that gets matched to the data by the Recipe System, not the individual primitives.

Technically, a primitive is a method of a primitive class. A primitive class gets associated with the input dataset by matching the AstroData tags. Once associated, all the primitives in that class, locally defined or inherited, are available to reduce that dataset. We refer to that collection of primitives as a "primitive set".

3.6 Primitive Set

A primitive set is a collection of primitives that are applicable to the input dataset. The association of the primitive set to the data is done by matching AstroData tags. It is a primitive set that gets passed to the recipe. The recipe can use any primitive within that set.

Technically, a primitive set is a class that can have inherited from other more general classes. In `geminidr`, there is a large inheritance tree of primitive classes from very generic to very specific. For example, the primitive set for GMOS images defines a few of its own primitives and inherits many other primitives from other sets (classes) like the one for generic CCD processing, the one related to photometry, the one that applies to all Gemini data, etc.

CHAPTER 4

The `reduce` command

4.1 Introduction

The `reduce` command is the DRAGONS Recipe System command line interface. The Recipe System also provides an application programming interface (API), whereby users and developers can programmatically invoke `Reduce` and set parameters on an instance of that class (see *The Reduce Class*).

Both interfaces allow users to configure and launch a Recipe System processing pipeline on one or more similar input datasets. Control of the Recipe System on the `reduce` command line is provided by a variety of options and switches which we will introduce in this chapter.

4.2 Usage Examples

Below we show examples that a user might typically want to do when using `reduce`. The command offers a lot of flexibility though, these examples are just a small subset of the possibilities. The objective here is to help the user get started.

4.2.1 Nominal usage

Because the Recipe System is automated, in many cases all that is needed is the command and a filename.

```
reduce S20161025S0111.fits
```

The system defaults to the "sq" mode, ie. science quality recipes. The best match recipe will be used with the best match primitive set. The required processed calibrations will be fetched from the local calibration manager.

The system defaults to using the Gemini Astrodata configuration package and the Gemini data reduction package, `gemini_instruments` and `geminidr`, respectively.

4.2.2 Overriding Primitive Parameters

The primitives for each set are given default values that have been found to give good results in most cases. Depending on the data and the science objectives, it might be necessary to tweak the primitive parameters to optimize the reduction. The `-p`, or in long form `--param` option allows the user to override the defaults.

```
reduce S20161025S0111.fits -p stackFrames:operation=median \
       stackFrames:reject_method=minmax
```

This sets the `stackFrames` input parameters `operation` and `reject_method` to `median` and `minmax`, respectively.

As one can see that, if several parameters are to be modified, the command can grow rather long. There is a way to keep it clean, see the section below on the *@file facility*.

4.2.3 Calling Specific Recipes and Primitives

The Recipe System's default behavior is to select the best recipe automatically. It is however possible, and sometimes required, to override this.

Override the default recipe

The first case where the recipe selection can be overridden is to select a recipe in the library different from the default. A recipe library can contain more than one recipe. Only one is set as the default. To let the Recipe System select the most appropriate recipe library, but then request the use of recipe within that library other than the default, simply state the name of the desired recipe. A good example is when making a bad pixel mask (BPM) for NIRI:

```
reduce @flats @darks -r makeProcessedBPM
```

Here the Recipes System will find the recipe library for NIRI flats (because the flats are first in the list), and then instead of running the default recipe which would in this case make a processed flat, it will run the `makeProcessedBPM` recipe.

For information about the `@` format, see *The @file Facility* below.

User recipe

It is possible for the user to force the use of a custom recipe. This is done with the `-r` flag again. The structure "recipe library containing recipes" must still be obeyed. Here is how the request is made:

```
reduce S20161025S0111.fits -r myrecipelibrary.myspecialrecipe
```

Both the name of the recipe library and, after the dot, the name of the recipe function are required. The path to the library can be prepended.

Calling a single primitive

Single primitives can be called directly from the command line bypassing the recipes entirely. A useful case is when one wants to display dataset. There is a primitive named `display`. The Recipe System will find the best-match primitive set, and then run the `display` primitive it contains.

```
reduce S20161025S0111.fits -r display
```

4.2.4 Manually Setting Calibrations

When the calibration manager is not available or if working on a new type of data not yet coded in the calibration association rules, it will be necessary to specify the processed calibration to use on the command line.

Another situation would be if one wanted to try various version of a calibration or different calibrations altogether to try to optimize a reduction. In such a case, one needs full control on which calibration is being used rather than always using the "best-match" returned by the local calibration manager.

```
reduce S20161025S0111.fits --user_cal processed_bias:S20161025S0200_bias.fits
```

4.3 Command Line Options and Switches

The `reduce` command help is provided by the `--help` option. This help is also available as a manual page as (`man reduce`). The options and switches are described further here.

4.3.1 Information 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.

The table provides a convenient view of all passed and default values for `reduce`. This can be useful when wanting to verify the syntax of a `reduce` call and to make sure everything has been parsed as expected.

Note that when not specified, *recipename* indicates 'None' because at this point in the execution the Recipe System has not yet been invoked and a default recipe not yet been determined. Eg.,

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

   Literals                      var 'dest'              Value
  ------------------------------------------------------------------
 ['-d', '--displayflags']        :: displayflags         :: True
 ['-p', '--param']               :: userparam            :: None
 ['--logmode']                   :: logmode              :: quiet
 ['--ql']                        :: mode                 :: sq
 ['--qa']                        :: mode                 :: sq
 ['--upload']                    :: upload               :: None
 ['-r', '--recipe']              :: recipename           :: None
 ['--adpkg']                     :: adpkg                :: None
 ['--suffix']                    :: suffix               :: None
 ['--drpkg']                     :: drpkg                :: geminidr
 ['--user_cal']                  :: user_cal             :: None
 ['--logfile']                   :: logfile              :: reduce.log
 ------------------------------------------------------------------

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

4.3.2 Configuration Switches and Options

–adpkg <ADPKG> Specify an external AstroData configuration package. This is used for non-Gemini instruments or during development of a new Gemini instrument. The package must be importable. The default AstroData

configuration package is `gemini_instruments` and it is distributed with DRAGONS.

E.g., `--adpkg scorpio_instruments`

–drpkg DRPKG Specify an external data reduction package. This is used for non-Gemini instruments or during development of a new Gemini instrument. The package must be importable. The default data reduction package is `geminidr` and it is distributed with DRAGONS.

E.g., `--drpkg scorpiodr`

–logfile <LOGFILE> Set the log file name. The default is `reduce.log` and it is written in the current directory.

–logmode <LOGMODE> Set logging mode. One of

- standard
- quiet
- debug

"quiet" writes only to the log file. The other modes writes information to the screen and to the log file. The default is "standard".

-p <USERPARAM [USERPARAM . . .]>, –param <USERPARAM [USERPARAM . . .]> Set a primitive input parameter from the command line. The form is

`-p primitivename:parametername=value`

This sets the parameter such that it applies only for the primitive "primitivename". To set multiple parameter-value pairs, separate them with whitespace, eg. `-p par1=val1 par2=val2`

The form `-p parametername=value` is also allowed but beware, that will sets any parameter with that name from any primitives to that value. It is somewhat dangerous and of limited use. It is to be seen as a global setting.

–qa Set the **mode** of operation to "qa", "quality assessment". When no "qa" or "ql" flag are specified the default mode is "sq". The "qa" mode is use internally at Gemini. Recipes differ depending on the mode.

–ql Set the **mode** of operation to "ql", "quicklook". When no "qa" or "ql" flag are specified the default mode is "sq". The "ql" mode is use for quick, near science quality reduction. Science quality is not guaranteed. Recipes differ depending on the mode. *This mode is not yet implemented. "ql" recipes are not yet available.*

-r <RECIPENAME>, –recipe <RECIPENAME> Specify a recipe by name. Users can request a non-default system recipe by names, e.g., `-r makeProcessedBPM`, or may specify their own recipe library and recipe function within. A user-defined recipe function must be "dotted" with the recipe file.

```
-r /path/to/recipes/recipelibrary.recipename
```

For a recipe file in the current working directory, the path can be omitted:

```
-r recipelibrary.recipename
```

A recipe library can contain more than one recipe. The recipe library must be a Python module, eg. `recipelibrary.py`. The recipes are Python functions within that module.

Finally, instead of specifying a recipe, it is possible to specify a primitive:

```
-r display
```

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

–upload Currently used internally (Gemini) only.

Send specific pipeline products to internal database. The default is None.

```
--upload metrics calibs
```

or equivalently:

```
--upload=metrics,calibs
```

–user_cal <USER_CAL [USER_CAL . . .]> Specify which processed calibration to use for the reduction. This override the selection from the local calibration manager. The syntax is:

```
--user_cal calibrationtype:path/calibrationfilename
```

Eg.:

```
--user_cal processed_bias:somepath/processed_bias.fits
```

The recognized calibration types are currently:

- processed_arc
- processed_bias
- processed_dark
- processed_flat
- processed_fringe
- processed_standard

4.4 The @file Facility

The reduce command line interface supports an "at-file" facility. An `@file` allows users to provide any and all command line options and flags to `reduce` in an acsii text file. This tool is very useful to keep the command line to a reasonable length and also to keep a record of the configurations that are applied. Here we illustrate how to use it.

4.4.1 Basic @file Usage

In a previous section we had an example where we were modifying a primmitive's input parameter values.

```
reduce S20161025S0111.fits -p stackFrames:operation=median \
       stackFrames:reject_method=minmax
```

Instead of typing the parameter settings on the command line, it might be more convenient to use an "at-file". We can write the parameter information in the "at-file" and add it to our `reduce` call. Let us have a file named "myreduction.par" with this content:

```
-p
stackFrames:operation=median
stackFrames:reject_method=minmax
```

Now we can call `reduce` as follow:

```
reduce S20161025S0111.fits @myreduction.par
```

By passing an `@file` to `reduce` on the command line, users can encapsulate all the options and positional arguments they may wish to specify in a single `@file`. It is possible to use multiple `@file` and even to embed one or more `@file` in another (see *Recursive @file Usage*). The parser opens all files sequentially and parses all arguments in the same manner as if they were specified on the command line.

To further illustrate the convenience provided by an `@file`, we'll continue with an example `reduce` command line that has even more arguments. We will also include new positional arguments, i.e., file names:

```
$ reduce -p stackFrames:operation=median stackFrames:reject_method=minmax \
  -r myrecipelib.myrecipe S20161025S0200.fits S20161025S0201.fits \
  S20161025S0202.fits S20161025S0203.fits S20161025S0204.fits
```

Here, two user parameters are being specified with `-p`, a recipe with `-r`, and a list of input datasets. We can write all this into a plain text `@file`, let's name it "reduce_args.par":

```
# input data files
S20161025S0200.fits
S20161025S0201.fits
S20161025S0202.fits
S20161025S0203.fits
S20161025S0204.fits

# primitive parameters optimization
--param

    # stackFrames
    stackFrames:operation=median
    stackFrames:reject_method=minmax

# recipe
-r
    myrecipelib.myrecipe
```

Now we can call `reduce` this way:

```
reduce @reduce_args.par
```

The order of the arguments in an `@file` is irrelevant, as is the file name. Also, the parser sees no difference across white space characters, such as space, tabs, newlines, etc. Comments are accommodated, both full line and in-line with the # character.

Finally, the "at-file" does not need to be in the current directory. A path can be given. For example:

```
reduce @../reduce_args.par
```

4.4.2 Recursive @file Usage

As implemented, the `@file` facility will recursively handle and process other `@file` specifications that appear in a `@file` or on the command line. For example, we may have another file containing a list of input files, let's call it "bias.lis":

```
# raw biases
S20161025S0200.fits
S20161025S0201.fits
S20161025S0202.fits
S20161025S0203.fits
S20161025S0204.fits
```

Then, we can add this list as an "at-file" in the `reduce_args.par` file:

```
# input files
@bias.lis

# primitive parameters optimization
--param

    # stackFrames
    stackFrames:operation=median
    stackFrames:reject_method=minmax

# recipe
-r
    myrecipelib.myrecipe
```

The `reduce` call becomes:

```
reduce @reduce_args.par
```

The parser will open and read the @bias.lis, 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.

Or one might want to keep the input files and the parameter settings separate. Then if we remove the `@bias.lis` from the "reduce_args.par" files, we can use it explicitly on the `reduce` command line:

```
reduce @bias.lis @reduce_args.par
```

4.4.3 Overriding @file Values

The `reduce` application employs a customized command line parser such that the command line option given in the `@file` can be modified on the command line *after* the `@file` has been processed.

The `-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. See Example 1 and 2 below.

For unitary value arguments, the command line value will *override* the `@file` value. See Example 3 below.

It is further specified that if one or more datasets (i.e. positional arguments) are passed on the command line, **all** files appearing as positional arguments in the "at-file" will be **replaced** by the one(s) on the command line. See Example 4 below.

In all cases, remember to use the `-d` option to verify the parsing if you are not sure.

Examples

The `@file` used in the examples, "reducepar", contains:

```
# input data files
S20161025S0200.fits
S20161025S0201.fits
S20161025S0202.fits
S20161025S0203.fits
S20161025S0204.fits
```

(continues on next page)

(continued from previous page)

```
# primitive parameters optimization
--param

    # stackFrames
    stackFrames:operation=median

# recipe
-r
    myrecipelib.myrecipe
```

Example 1: Accumulate a new parameter:

```
reduce @reducepar --param stackFrames:hsigma=5.0

Summary of parsed options:
--------------------------
Input files: no changes
Parameters: ['stackFrames:operation=median', 'stackFrames:hsigma=5.0']
Recipe: no changes
```

Example 2: Override a parameter defined in the `@file`:

```
reduce @reducepar --param stackFrames:operation=wtmean

Summary of parsed options:
--------------------------
Input files: no changes
Parameters: ['stackFrames:operation=wtmean']
Recipe: no changes
```

Example 3: Override the recipe:

```
reduce @reducepar -r myrecipelib.different_recipe

Summary of parsed options:
--------------------------
Input files: no changes
Parameters: no changes
Recipe: myrecipelib.different_recipe
```

Example 4: Override the input files. All the files in the `@files` will be ignored:

```
reduce @reducepar S20161025S0111.fits

Summary of parsed options:
--------------------------
Input files: S20161025S0111.fits
Parameters: no changes
Recipe: no changes
```

CHAPTER 5

The Reduce Class

The Reduce class provides the underlying structure of the reduce command. This section describes and discusses the programmatic interface available on the class Reduce. This section is for users wanting to use the Reduce class programmatically.

The reduce application introduced in the previous chapter is a user interface script providing a command line access to the Reduce class. The reduce application parses the arguments and initialize the Reduce class and its runr method. It is possible to bypass the reduce command and sets attributes directly on an instance of Reduce, as the following discussion illustrates.

5.1 Using Reduce

The Reduce class is defined in the recipe_system.reduction.coreReduce module. The Reduce class provides a set of attributes and one public method, runr that launches a reduction. This is the only public method on the class.

5.1.1 Very Basic Usage

The most basic usage involves importing the class, instantiating it, assigning a file to reduce and then launching the runr method.

```
>>> from recipe_system.reduction.coreReduce import Reduce
>>> myreduce = Reduce()
>>> myreduce.files.append('S20161025S0111.fits')
>>> myreduce.runr()
```

5.1.2 Typical Usage for Reduction

A more typical usage for reducing data can involve setting other options and can include setting up a *logger*. When using the Gemini data reduction primitives, the logger is highly recommended.

Normal usage will also likely involve the use of the calibration database facility, `caldb`. We will ignore `caldb` here and rather fully describe it and its usage in a subsequent chapter, *Local Calibration Database*. See api_example where we put it all together.

```
>>> from recipe_system.reduction.coreReduce import Reduce
>>> from gempy.utils import logutils
>>>
>>> logutils.config(file_name='example.log')
>>>
>>> inputfiles = ['S20161025S0200.fits', 'S20161025S0201.fits']
>>> myreduce = Reduce()
>>> myreduce.files = inputfiles
>>> myreduce.runr()
```

Neither `coreReduce` nor the `Reduce` class initializes any logging activity. This is the responsibility of the programmer. The Recipe System does not require a logger but the Gemini primitives do. The absence of a logger when using the Gemini data reduction package leads to double the reporting on the screen. More an annoyance than a problem, admittedly.

You are free to provide your own logger, or you can use the fully defined logger provided in DRAGONS. It is recommended that you use the system logger as `Reduce` is tuned to use the DRAGONS logger.

Returning to the example above, we could also set the recipe to a custom recipe, override a primitive parameters, set a data reduction package, etc. The attributes that can be set are discussed in *Public Attributes to Reduce* below.

```
>>> myreduce.recipename = 'myrecipelib.myrecipe'
>>> myreduce.uparms = [('stackFrames:operation', 'median')]
>>> myreduce.dkpkg = 'thirdpartydr'
>>> # rerun with the modified recipe and parameter
>>> myreduce.runr()
```

A notable quirk is how to set the `adpkg` that is defined in the `reduce` command line interface. The `Reduce` class does not have an attribute for it. Instead, the programmer must import any third party AstroData instrument configuration files explicitely *before* launching `runr`.

```
>>> import astrodata
>>> import thirdparty_instruments
>>>
>>> myreduce.Reduce()
>>> myreduce.drpkg = 'thirdpartydr'
>>> myreduce.files.append('filename.fits')
>>> myreduce.runr()
```

5.2 Public Attributes to `Reduce`

Public Attribute	Python type	Default
files	<type 'list' of 'str'>	[]
output_filenames	<type 'list' of 'str'>	None
mode	<type 'str'>	'sq'
recipename	<type 'str'>	'_default'
drpkg	<type 'str'>	'geminidr'
suffix	<type 'str'>	None
ucals	<type 'dict'>	None
uparms	<type 'list' of 'tuple'>	None
upload	<type 'list' of 'str'>	None

files A list of input file names to reduce. Only the first file in the list will be used for the recipe and primitive selection.

```
myreduce.files.extend(['S20161025S0200.fits', 'S20161025S0201.fits'])
```

output_filenames A list of output file names. This **cannot** be set. It is a return value. It is used *after* the recipe has run to collect the names of the files that were created.

```
output_stack = myreduce.output_filenames[0]
```

mode The reduction mode. The Gemini data reduction package currently supports 'sq' and 'qa', with 'ql' in the works. ['sq': Science Quality, 'qa': Quality Assessment, 'ql': Quick Look Reduction.]

```
myreduce.mode = 'qa'
```

recipename The name of the recipe to use. If left to "_default", the Recipe System will invoke the mappers and select the best matching recipe library and use its default recipe.

If only the name of a recipe is provided, the mappers will be invoked to find the best matching recipe library and use the named recipe rather than the default.

If a "module.recipe" string is provided, the user's "module" will be imported and the user's "recipe" will be used. No mapping will be done.

```
myreduce.recipename = 'myrecipelib.myrecipe'
```

If the name of a primitive is given, the Recipe System will find the best match primitive set and run the specified primitives from that set.

suffix The suffix to add the final outputs of a recipe. In the Gemini primitives, default suffixes are assigned to each primitives. Setting `suffix` will override the default suffix of the last primitive in the recipe.

```
myreduce.suffix = '_flatBfilter'
```

drpkg The name of the data reduction package to use. The default is `geminidr`. If using a third-party package, or during new instrument development, set this attributes to import the correct suite of recipes and primitives.

```
myreduce.drpkg = 'scorpiodr'
```

ucals Set the processed calibration to be used. This overrides the automatic selection done by the calibration manager, if one is being used. This setting must be used if no calibration manager is used or available, or when, for example, the calibrations association rules are not yet implemented. It is also useful for testing and for getting full control of the calibrations being used.

The format for this attribute's value is somewhat complicated. It is recommended to use the `normalize_ucals` function in the `recipe_system.utils.reduce_utils` module to get the dictionary this attribute expects.

The format needs to looks like this:

```
{(ad.calibration_key(), 'processed_bias'): '/path/master_bias.fits'}
```

There must be one entry per input files for each type of calibrations.

The recognized calibration types are currently:

- processed_arc
- processed_bias
- processed_dark
- processed_flat
- processed_fringe
- processed_standard

Here's how to use `normalize_ucals`:

```
from recipe_system.utils.reduce_utils import normalize_ucals

mycalibrations = ['processed_bias:/path/master_bias.fits',
                  'processed_flat:/path/master_Bflat.fits']

ucals_dict = normalize_ucals(myreduce.files, mycalibrations)
myreduce.ucals = ucals_dict
```

uparms Set primitive parameter values. This will override the primitive defaults. This is a list of tuples with the primitive name and parameter in the first element, and the value in the second one.

```
myreduce.uparms = [('stackFrames:operation', 'median')]
```

If the primitive name is omitted all parameters with that name, in any primitives will be reset. Be careful.

upload **Internal use only**. Specify which types of product to upload to the Gemini internal database. Allowed values are "metrics", "calibs", and "science", the latter is planned but not yet implemented.

CHAPTER 6

Local Calibration Database

The Recipe System has a system to retrieve processed calibration automatically. This system must work with a Calibration Manager. Currently, only one public Calibration Manager is available, the Gemini Calibration Manager, `GeminiCalMgr`. This must be installed as a DRAGONS dependency; a conda install will take care of that (see *Installation*).

The Calibration Manager contains the calibration association rules and database access hooks. The Gemini Calibration Manager uses exactly the same calibration association rules as the Gemini Observatory Archive (GOA).

The calibration facility requires a database. The Recipe System's `caldb` application helps the user configure and create a local, lightweight `sqlite` database, and add or remove calibration files to and from that database.

In this chapter, we explain how to use `caldb` to add processed calibrations that the Recipe System will pick up when needed.

Note: We intend to improve the Calibration Manager side of things to make expanding the association rules for new instruments or non-Gemini instruments feasible.

6.1 Configuring `caldb`

The first time `caldb` is used for a project, either via command line or API, it needs to be configured and initialized. The configuration is stored in a text file in a special directory named `~/.geminidr/`, in a file called `rsys.cfg`. The `~` means the user's home directory. The very first step, to be done only once, is to create the directory and the configuration file.

```
$ mkdir ~/.geminidr
$ touch ~/.geminidr/rsys.cfg
```

The `rsys.cfg` file must contain the following lines:

```
[calibs]
standalone = True
database_dir = ~/.geminidr    # set this path to whatever you want.
```

The `standalone` option tells `caldb` if you are using a local database when it is set to True. `standalone = False` is used only internally at Gemini when using the internal data manager.

The `database_dir` parameter points to the directory hosting the calibration database. The database name is always `cal_manager.db`, this cannot be set, only the directory where it lives. It is possible to have more than one database as long as they are in different directory. Which one will be picked up will be set through the `database_dir` parameter in `rsys.cfg`.

6.2 Using `caldb` on the Command Line

The `caldb` tool is used to interact with the local calibration database. This is where the Recipe System will look for processed calibrations. For a reminder of its basic usage, one can always use the `--help` flag:

```
$ caldb --help
usage: caldb [-h] {config,init,list,add,remove} ...

Calibration Database Management Tool

positional arguments:
  {config,init,list,add,remove}
                        Sub-command help
    config              Display configuration info
    init                Create and initialize a new database.
    list                List calib files in the current database.
    add                 Add files to the calibration database. One or more
                        files or directories may be specified.
    remove              Remove files from the calibration database. One or
                        more files may be specified.

optional arguments:
  -h, --help            show this help message and exit
```

There can be only one positional argument given to `caldb`, this means only one file at a time can be added or removed from the database.

Once the configuration file is in place (see *Configuring caldb*), one can verify the configuration by doing:

```
$ caldb config

Using configuration file: ~/.geminidr/rsys.cfg
Active database directory: /Users/username/.geminidr
Database file: /Users/username/.geminidr/cal_manager.db

The 'standalone' flag is active, meaning that local calibrations will be used
```

To initialize a new database with the selected configuration:

```
$ caldb init
```

Once the database is initialized (created), it is ready for use.

To add a file:

```
$ caldb add /path/to/master_bias.fits
```

If the path is not given, the current directory is assumed. The addition of a file to the database is simply the addition of the filename and its location on the disk. The file itself *is not stored*. If the calibration file is deleted or moved, the database will not know and still think that the file is there.

To see what is in the database:

```
$ caldb list
master_bias.fits    /path/to/
```

To remove a file from the database:

```
$ caldb remove master_bias.fits
```

Warning: If a file that is already stored within the database needs updating, it will need to be removed and added again. `caldb` has no update tool.

To see `caldb` used in a complete example along with the other tools see commandline_example.

6.3 Using the `caldb` API

Before being usable in a Python program, the local calibration manager must be configured. This cannot be done from the API. See *Configuring caldb* for instructions.

The calibration database is initialized and the configuration are read into the the calibration service as follow:

```
>>> from recipe_system import cal_service
>>>
>>> caldb = cal_service.CalibrationService()
>>> caldb.config()
>>> caldb.init()
>>> cal_service.set_calservice()
```

The calibration service is then ready to use. This must be done before `Reduce` is instantiated.

To add a processed calibration to the database:

```
>>> caldb.add_cal('/path/to/master_bias.fits')
```

If the path is not given, the current directory is assumed. The addition of a file to the database is simply the addition of the filename and its location on the disk. The file itself *is not stored*. If the calibration file is deleted or moved, the database will not know and still think that the file is there.

To see what is in the database:

```
>>> for f in caldb.list_files():
...     print(f)
...
FileData(name=u'master_bias.fits', path=u'/path/to')
```

To remove a file from the database:

```
>>> caldb.remove_cal('master_bias.fits')
```

Warning: If a file that is already stored within the database needs updating, it will need to be removed and added again. `caldb` has no update tool.

To see it used in a complete example along with the other tools see api_example.

CHAPTER 7

Supplemental tools

DRAGONS provides a number of command line tools that users should find helpful in executing `reduce` on their data. Some of those tools also offer an API.

These supplemental tools can help users discover information, not only about their own data, but about the Recipe System, such as available recipes, primitives, and defined tags.

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

7.1 dataselect

The tool `dataselect` will help with the bookkeeping and with creating lists of input files to feed to the Recipe System. The tool has a command line and an API. This tool finds files that match certain criteria defined through AstroData Tags and expressions involving AstroData Descriptors.

You can access the basic documentation from the command line by typing:

```
$ dataselect --help

usage: dataselect [-h] [--tags TAGS] [--xtags XTAGS] [--expr EXPRESSION]
                  [--strict] [--output OUTPUT] [--verbose] [--debug]
                  inputs [inputs ...]

Find files that matches certain criteria defined by tags and expression
involving descriptors.

positional arguments:
  inputs                Input FITS file

optional arguments:
  -h, --help            show this help message and exit
  --tags TAGS, -t TAGS  Comma-separated list of required tags.
  --xtags XTAGS         Comma-separated list of tags to exclude
  --expr EXPRESSION     Expression to apply to descriptors (and tags)
```

(continues on next page)

(continued from previous page)

```
  --strict              Toggle on strict expression matching for exposure_time
                        (not just close) and for filter_name (match component
                        number).
  --output OUTPUT, -o OUTPUT
                        Name of the output file
  --verbose, -v         Toggle verbose mode when using -o
  --debug               Toggle debug mode
```

7.1.1 `dataselect` Command Line Tool

`dataselect` accepts list of input files separated by space, and wildcards. Below are some usage examples.

1. This command selects all the FITS files inside the `raw` directory with a tag that matches `DARK`.

```
$ dataselect raw/*.fits --tags DARK
```

2. To select darks of a specific exposure time:

```
$ dataselect raw/*.fits --tags DARK --expr='exposure_time==20'
```

3. To send that list to a file that can be used later:

```
$ dataselect raw/*.fits --tags DARK --expr='exposure_time==20' -o dark20s.lis
```

4. This commands prints all the files in the current directory that *do not* have the `CAL` tag (calibration files).

```
$ dataselect raw/*.fits --xtags CAL
```

5. The `xtags` can be used with `tags`. To select images that are not flats:

```
$ dataselect raw/*.fits --tags IMAGE --xtags FLAT
```

6. This command selects all the files with a specific target name:

```
$ dataselect --expr 'object=="FS 17"' raw/*.fits
```

7. This command selects all the files with an "observation_class" descriptor that matches the "science" value and a specific exposure time:

```
$ dataselect --expr '(observation_class=="science" and exposure_time==60.)' raw/*.
↪fits
```

7.1.2 `dataselect` API

The same selections presented in the command line section above can be done from the `dataselect` API. Here is the API versions of the examples presented in the previous sections.

The list of files on disk must first be obtained with Python's `glob` module.

```
>>> import glob
>>> all_files = glob.glob('raw/*.fits')
```

The `dataselect` module is located in `gempy.adlibrary` and must first be imported:

```
>>> from gempy.adlibrary import dataselect
```

1. This command selects all the FITS files inside the `raw` directory with a tag that matches `DARK`.

   ```
   >>> all_darks = dataselect.select_data(all_files, ['DARK'])
   ```

2. To select darks of a specific exposure time:

   ```
   >>> expression = 'exposure_time==20'
   >>> parsed_expr = dataselect.expr_parser(expression)
   >>> darks20 = dataselect.select_data(all_files, ['DARK'], [], parsed_expr)
   ```

3. To send that list to a file that can be used later:

   ```
   >>> expression = 'exposure_time==20'
   >>> parsed_expr = dataselect.expr_parser(expression)
   >>> darks20 = dataselect.select_data(all_files, ['DARK'], [], parsed_expr)
   >>> with open('dark20s.lis', 'w') as f:
   ...     for filename in dark20:
   ...         f.write(filename + '\n')
   ...
   >>>
   ```

   Note that the need to send a list of a file on disk will probably not be very common when using the API as `Reduce` will take the Python list directly.

4. This commands prints all the files in the current directory that *do not* have the `CAL` tag (calibration files).

   ```
   >>> non_cals = dataselect.select_data(all_files, [], ['CAL'])
   ```

5. The `xtags` can be used with `tags`. To select images that are not flats:

   ```
   >>> has_tags = ['IMAGE']
   >>> has_not_tags = ['FLAT']
   >>> non_flat_images = dataselect.select_data(all_files, has_tags, has_not_tags)
   ```

6. This command selects all the files with a specific target name:

   ```
   >>> expression = 'object="FS 17"'
   >>> parsed_expr = dataselect.expr_parser(expression)
   >>> stds = dataselect.select_data(all_files, expression=parsed_expr)
   ```

7. This command selects all the files with an "observation_class" descriptor that matches the "science" value and a specific exposure time:

   ```
   >>> expression = '(observation_class=="science" and exposure_time==60.)'
   >>> parsed_expr = dataselect.expr_parser(expression)
   >>> sci60 = dataselect.select_data(all_files, expression=parsed_expr)
   ```

7.1.3 The `strict` Flag

The `strict` flag applies to the descriptors `exposure_time()` and `filter_name()`. To keep the user interface more friendly, in the expressions, the exposure time is matched on a "close enough" principle and the filter name is matched on a "general bandpass name" principle.

For example, if the exposure time in the header is 10.001 second, from a user's perspective, asking to match "10" seconds is a lot nicer, `exposure_time==10`. Similarly, asking for the "H"-band filter is more natural than asking for the "H_G0203" filter.

However, there might be cases where the exposure time or the filter name must be matched *exactly*. In such case, the `strict` flag should be activated. For example:

```
$ dataselect raw/*.fits --strict --expr='exposure_time==0.95'
```

And:

```
>>> expression = 'exposure_time==0.95'
>>> parsed_expr = dataselect.expr_parser(expression, strict=True)
>>> filelist = dataselect.select_data(all_files, expression=parsed_expr)
```

7.2 showd

The `showd` command line tool helps the user gather information about files on disk. The "d" in `showd` stands for "descriptor". `showd` is used to show the value of specific AstroData descriptors for the files requested.

Its basic usage can be printed using the following command:

```
$ showd --help
usage: showd [-h] --descriptors DESCRIPTORS [--csv] [--debug]
             [inputs [inputs ...]]

For each input file, show the value of the specified descriptors.

positional arguments:
  inputs                Input FITS files

optional arguments:
  -h, --help            show this help message and exit
  --descriptors DESCRIPTORS, -d DESCRIPTORS
                        comma-separated list of descriptor values to return
  --csv                 Format as CSV list.
  --debug               Toggle debug mode
```

One or more descriptors can be printed together. Here is an example::

```
$ showd -d object,exposure_time *.fits
-------------------------------------------
filename                object   exposure_time
-------------------------------------------
N20160102S0275.fits    SN2014J          20.002
N20160102S0276.fits    SN2014J          20.002
N20160102S0277.fits    SN2014J          20.002
N20160102S0278.fits    SN2014J          20.002
N20160102S0279.fits    SN2014J          20.002
N20160102S0295.fits      FS 17          10.005
N20160102S0296.fits      FS 17          10.005
N20160102S0297.fits      FS 17          10.005
N20160102S0298.fits      FS 17          10.005
N20160102S0299.fits      FS 17          10.005
```

Above is a human-readable table. It is possible to return a comma-separated list, CSV list, with the `--csv` tag:

```
$ showd -d object,exposure_time *.fits --csv
filename,object,exposure_time
N20160102S0275.fits,SN2014J,20.002
N20160102S0276.fits,SN2014J,20.002
N20160102S0277.fits,SN2014J,20.002
N20160102S0278.fits,SN2014J,20.002
N20160102S0279.fits,SN2014J,20.002
N20160102S0295.fits,FS 17,10.005
N20160102S0296.fits,FS 17,10.005
N20160102S0297.fits,FS 17,10.005
N20160102S0298.fits,FS 17,10.005
N20160102S0299.fits,FS 17,10.005
```

The `showd` command also integrates well with `dataselect`. You can use dataselect together with `showd` if you want to print the descriptors values in a data subset:

```
$ dataselect raw/*.fits --tag FLAT | showd -d object,exposure_time
------------------------------------------------
filename                 object   exposure_time
------------------------------------------------
N20160102S0363.fits   GCALflat          42.001
N20160102S0364.fits   GCALflat          42.001
N20160102S0365.fits   GCALflat          42.001
N20160102S0366.fits   GCALflat          42.001
N20160102S0367.fits   GCALflat          42.001
```

The "pipe" ' | ' gets the `dataselect` output and passes it to `showd`.

7.3 showrecipes

The Recipe System will select the best recipe for your data, which can be overriden when necessary. To see what sequence of primitives a recipe will execute or which recipes are available for the dataset, one can use `showrecipes`.

7.3.1 Show Recipe Content

To see the content of the best-matched default recipes:

```
$ showrecipes S20170505S0073.fits
```

```
Recipe not provided, default recipe (makeProcessedFlat) will be used.
Input file: /path_to/S20170505S0073.fits
Input tags: ['FLAT', 'LAMPOFF', 'AZEL_TARGET', 'IMAGE', 'DOMEFLAT',
'GSAOI', 'RAW', 'GEMINI', 'NON_SIDEREAL', 'CAL', 'UNPREPARED', 'SOUTH']
Input mode: sq
Input recipe: makeProcessedFlat
Matched recipe: geminidr.gsaoi.recipes.sq.recipes_FLAT_IMAGE::makeProcessedFlat
Recipe location: /path_to/dragons/geminidr/gsaoi/recipes/sq/recipes_FLAT_IMAGE.py
Recipe tags: set(['FLAT', 'IMAGE', 'GSAOI', 'CAL'])
Primitives used:
   p.prepare()
   p.addDQ()
   p.nonlinearityCorrect()
   p.ADUToElectrons()
```

(continues on next page)

(continued from previous page)

```
   p.addVAR(read_noise=True, poisson_noise=True)
   p.makeLampFlat()
   p.normalizeFlat()
   p.thresholdFlatfield()
   p.storeProcessedFlat()
```

To see the content of a specific recipe:

```
$ showrecipes S20170505S0073.fits -r makeProcessedBPM
```

```
Input file: /path_to/S20170505S0073.fits
Input tags: ['FLAT', 'LAMPOFF', 'AZEL_TARGET', 'IMAGE', 'DOMEFLAT',
'GSAOI', 'RAW', 'GEMINI', 'NON_SIDEREAL', 'CAL', 'UNPREPARED', 'SOUTH']
Input mode: sq
Input recipe: makeProcessedBPM
Matched recipe: geminidr.gsaoi.recipes.sq.recipes_FLAT_IMAGE::makeProcessedBPM
Recipe location: /path_to/dragons/geminidr/gsaoi/recipes/sq/recipes_FLAT_IMAGE.pyc
Recipe tags: set(['FLAT', 'IMAGE', 'GSAOI', 'CAL'])
Primitives used:
   p.prepare()
   p.addDQ()
   p.addVAR(read_noise=True, poisson_noise=True)
   p.ADUToElectrons()
   p.selectFromInputs(tags="DARK", outstream="darks")
   p.selectFromInputs(tags="FLAT")
   p.stackFrames(stream="darks")
   p.makeLampFlat()
   p.normalizeFlat()
   p.makeBPM()
```

7.3.2 Show Index of Available Recipes

Of course in order to ask for a specific recipe, it is useful to know which recipes are available to the dataset. To see the index of available recipes:

```
$ showrecipes S20170505S0073.fits --all
```

```
Input file: /path_to/S20170505S0073.fits
Input tags: set(['FLAT', 'LAMPOFF', 'AZEL_TARGET', 'IMAGE', 'DOMEFLAT',
'GSAOI', 'RAW', 'GEMINI', 'NON_SIDEREAL', 'CAL', 'UNPREPARED', 'SOUTH'])
Recipes available for the input file:
   geminidr.gsaoi.recipes.sq.recipes_FLAT_IMAGE::makeProcessedBPM
   geminidr.gsaoi.recipes.sq.recipes_FLAT_IMAGE::makeProcessedFlat
   geminidr.gsaoi.recipes.qa.recipes_FLAT_IMAGE::makeProcessedFlat
```

The output shows that there are two recipes for the SQ (Science Quality) mode and one recipe for the QA (Quality Assesment) mode. By default, the Recipe System uses the SQ mode for processing the data.

As for the other commands, you can use the `--help` or `-h` flags on the command line to display the help message.

7.4 showpars

The `showpars` application is a simple command line utility allowing users to see the available parameters and defaults for a particular primitive function applicable to a given dataset. Since the applicable primitives for a dataset are dependent upon the *tagset* of the identified dataset (i.e. `NIRI IMAGE`, `F2 SPECT`, `GMOS BIAS`, etc.), which is to say, the *kind* of data we are looking at, the parameters available on a named primitive function can vary across data types, as can the primitive function itself. For example, F2 IMAGE `stackFlats` uses the generic implementation of the function, while GMOS IMAGE `stackFlats` overrides that generic method.

We examine the help on the command line of showpars:

```
$ showpars -h
usage: showpars [-h] [-v] filen primn

Primitive parameter display, v2.2.0

positional arguments:
  filen          filename
  primn          primitive name

optional arguments:
  -h, --help     show this help message and exit
  -v, --version  show program's version number and exit
```

Two arguments are required: the dataset filename, and the primitive name of interest. As readers will note, `showpars` provides a wealth of information about the available parameters on the specified primitive, including allowable values or ranges of values:

```
$ showpars S20180516S0237.fits stackFlats
Dataset tagged as set(['RAW', 'GMOS', 'GEMINI', 'SIDEREAL', 'FLAT',
'UNPREPARED', 'IMAGE', 'CAL', 'TWILIGHT', 'SOUTH'])
Settable parameters on 'stackFlats':
=====================================
 Name                       Current setting

suffix               '_stack'            Filename suffix
apply_dq             True                Use DQ to mask bad pixels?
scale                False               Scale images to the same intensity?
operation            'mean'              Averaging operation
Allowed values:
    wtmean  variance-weighted mean
    mean    arithmetic mean
    median  median
    lmedian low-median

reject_method        'minmax'            Pixel rejection method
Allowed values:
    minmax  reject highest and lowest pixels
    none    no rejection
    varclip reject pixels based on variance array
    sigclip reject pixels based on scatter

hsigma               3.0                 High rejection threshold (sigma)
    Valid Range = [0,inf)
lsigma               3.0                 Low rejection threshold (sigma)
    Valid Range = [0,inf)
mclip                True                Use median for sigma-clipping?
```

(continues on next page)

(continued from previous page)

```
max_iters             None                  Maximum number of clipping iterations
    Valid Range = [1,inf)
nlow                  0                     Number of low pixels to reject
    Valid Range = [0,inf)
nhigh                 0                     Number of high pixels to reject
    Valid Range = [0,inf)
memory                None                  Memory available for stacking (GB)
    Valid Range = [0.1,inf)
```

With this information, users can adjust parameters for particular primitive functions. As we have seen already, this can be done from the `reduce` command line or the `Reduce` class. Building on material covered in this manual, and continuing our example from above::

```
$ reduce -p stackFlats:nhigh=3 <fitsfiles> [ <fitsfile>, ... ]
```

And the reduction proceeds. When the `stackFlats` primitive begins, the new value for `nhigh` will be used.

Note: Advanced User. Inheritance and class overrides within the primitive and parameter hierarchies means that one cannot simply look at any given primitive function and its parameters and extrapolate those to all such named primitives and parameters. Primitives and their parameters are tied to the particular classes designed for those datasets identified as a particular kind of data.

7.5 typewalk

The `typewalk` application examines files in a directory or directory tree and reports the data classifications through the `astrodata` tag sets. By default, typewalk will recurse all subdirectories under the current directory. Users may specify an explicit directory with the `-d, --dir` option.

`typewalk` supports the following options:

```
-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 tags. 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 'tags' criteria. If not specified,
                      matching logic is AND (See --tags). Eg., --or --tags
                      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.
```

(continues on next page)

(continued from previous page)

```
--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 **–tags** option indicates, `typewalk` can find and report data that match specific tag criteria. For example, a user might want to find all GMOS image flats (`--tags GMOS IMAGE FLAT`) under a certain directory. `typewalk` will locate and report all datasets that would match the AstroData tags, `set(['GMOS', 'IMAGE', 'FLAT'])`.

A user may request that an output file be written containing all datasets matching AstroData tag qualifiers passed by the **–tags** 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'. However, for such use, dataselect is probably preferable as it is more versatile than `typewalk`.

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

As a simple example, find all F2 SPECT datasets in a directory tree:

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

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

For example, find GMOS, IMAGE tag sets, 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)
```

CHAPTER 8

Acknowledgments

The Gemini Observatory is operated by the Association of Universities for Research in Astronomy, Inc., under a cooperative agreement with the NSF on behalf of the Gemini partnership: the National Science Foundation (United States), the National Research Council (Canada), CONICYT (Chile), Ministerio de Ciencia, Tecnología e Innovación Productiva (Argentina), Ministério da Ciência, Tecnologia e Inovação (Brazil), and Korea Astronomy and Space Science Institute (Republic of Korea).

APPENDIX A

Glossary

astrodata Package distributed with the DRAGONS meta-package. `astrodata` is used to open datasets and provide an uniform interface to the data and the metadata (eg. headers) regardless of whether the file on disk is a FITS file or some other format, whether it is a GMOS file or NIRI file. The Recipe System relies critically on `astrodata`.

AstroData Not to be confused with `astrodata`, this is the base class for instrument-specific AstroData classes, and the one most users and developers will interact with at a programmatic level.

descriptor A descriptor is a high-level access to essential dataset metadata (eg. headers) through a uniform, instrument-independent interface. E.g., `ad.gain()`. A descriptor is a method on an `AstroData` instance.

DRAGONS Data Reduction for Astronomy from Gemini Observatory North and South.

A suite of packages comprising `astrodata`, `gemini_instruments`, the `recipe_system`, `geminidr`, and `gempy`, which together provide the full functionality needed to run recipe pipelines on observational datasets. DRAGONS can be referred to as a framework.

gempy A DRAGONS package comprising various functional utilities, some generic, some Gemini-specific.

primitive A function defined within a data reduction instrument package that performs actual work on a dataset. Primitives observe 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, and must have the same set of input parameters. This is a Gemini Coding Standard; it is not enforced by the Recipe System.

recipe A function defined in a recipe library (module) which defines a sequence of calls to primitives. A recipe is a simple python function that receives an instance of the appropriate primitive class (primitive set) and executes the primitive sequence defined in the recipe function. Users can pass recipe names directly to `reduce`.

Recipe System The DRAGONS framework that automates the selection and execution of recipes and primitives. The Recipe System defines a set of classes that uses attributes on an astrodata instance to locate recipes and primitives appropriate to the dataset.

reduce The command line interface to the Recipe System.

tags [or tagset] Represents a data classification. When loaded with `AstroData`, a dataset will be classified with a number of tags that describe both the data and its processing state. The tags are defined in *astrodata packages*,

eg. the Gemini package is `gemini_instruments`.

Todo: Update here for Anaconda/astroconda.

(The *original entry* is located in /home/docs/checkouts/readthedocs.org/user_builds/dragons-recipe-system-users-manual/checkouts/v3.0.3/recipe_system/doc/rs_UsersManual/notused/appendix_demo.rst, line 12.)

Todo: The new recipe libraries have no `reduceDemo` recipe.

(The *original entry* is located in /home/docs/checkouts/readthedocs.org/user_builds/dragons-recipe-system-users-manual/checkouts/v3.0.3/recipe_system/doc/rs_UsersManual/notused/appendix_demo.rst, line 125.)

Todo: What about remote database?

(The *original entry* is located in /home/docs/checkouts/readthedocs.org/user_builds/dragons-recipe-system-users-manual/checkouts/v3.0.3/recipe_system/doc/rs_UsersManual/notused/caldb.rst, line 49.)

Index