DRAGONS - Recipe System Programmer's Manual

Release 3.0.4

Kenneth Anderson, Kathleen Labrie, Bruno Quint

November 2022

Contents

1 Introduction 2

2 Overview 4

3 The Mappers 10

4 Using the Mappers API 14

5 Acknowledgments 22

Appendices

A Data Reduction Packages 23

B JIT Calibration Request 27

C Glossary 28

Index 30

Document ID

PIPE-USER-108_RSProgManual

CHAPTER 1

Introduction

This document is the Programmer Manual for DRAGONS Recipe System. It presents detailed information and discussion about the programmatic interfaces on the system's underlying classes, `RecipeMapper` and `PrimitiveMapper`. It describes usage of the Recipe System's application programming interface (API).

Subsequent to this introduction, this document provides a high-level overview of the Recipe System (*Chapter 2, Overview*), which is then followed by an introduction to the mapper classes in *Chapter 3, The Mappers*. It then presents the interfaces on the mapper classes and how to use them to retrieve the appropriate recipes and primitives.

It also provides instruction on how to write recipes the Recipe System can recognise and use, how one can interactively run a recipe, step by step, perform discovery on the data processed, and pass adjusted parameters to the next step of a recipe.

Details and information about Astrodata 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 the Recipe System.

1.1 Reference Documents

-

1.2 Related Documents

-
-

1.3 Further Information

As this document details programmatic use of the mapper classes, readers who wish to read up on the Recipe System application, , should instead consult the DRAGONS document, , which also describes usage of the `Reduce` class API

from a user's point of view rather than a programmer's.

Users and developers wishing to see more information about the Astrodata package, how to use the programmatic interfaces on such objects should consult the documents *listed above*.

```
>>> ad = astrodata.open(<fitsfile>)
|   >>> dtags = set(list(ad.tags)[:])
|   >>> instpkg = ad.instrument(generic=True).lower()
|   >>> pm = PrimitiveMapper(dtags, instpkg)
|   >>> pclass = pm.get_applicable_primitives()
|   >>> pclass
```

CHAPTER 2

Overview

The Recipe System is a python package distributed as a component of the Gemini Observatory's DRAGONS meta-package for data reduction. The Recipe System is a framework that supports configurable data processing pipelines and which can technically accommodate processing pipelines for arbitrary types of astronomical data.

The Recipe System provides infrastructure that will inspect DRAGONS-compatible configuration and data processing packages and map recipes and primitives to the input data. The infrastructure will then link the appropriate primitives to the appropriate recipe and run it. The Recipe System also includes tools and mechanisms to automatically handle and associate processed calibrations using the `GeminiCalMgr` python package, available separately.

The Recipe System matches and links recipes and primitives to the data, and execute the recipes. Any flow control and decisions are left to the recipes and primitives. The Recipe System only provides an automation infrastructure to run the reduction.

To match data to recipes and primitives, the Recipe System requires the data to be accessed with `astrodata`, the DRAGONS data access infrastructure. `astrodata` provides a common grammar to recognize and to access pixel and header data. The Recipe System depends critically on `astrodata` for recipe and primitive mapping which matches `astrodata` tags. (See the *Astrodata documentation, Section 1.2* for more information.)

Gemini Observatory has developed a suite of recipes and primitives for the Observatory's facility instruments. These can be found in the DRAGONS package `geminidr`. The Astrodata configuration package for the Gemini instruments is `gemini_instruments`. Both packages are included in DRAGONS.

At Gemini, DRAGONS and its Recipe System are used in an operational environment for data quality assessment at night. It provides sky condition metrics calculated from the data itself. DRAGONS and the Recipe System also are the data reduction platform that will replace Gemini's legacy IRAF software.

Readers unfamiliar with terms and concepts presented in this manual can consult the *Glossary* for a definition of terms. For greater detail and depth, below are definitions of terms and components directly relevant to usage and development of the Recipe System.

2.1 Definitions

The following section provides definitions, discussion, and some examples about certain terms and components that are key to the functioning Recipe System.

Data Reduction Package (''drpkg'')

A data reduction package is a Python package in which data reduction software, formatted as recipes and primitives, is organized. Normally, recipes and primitives specific to an instrument are organized together in an "instrument package". The primitives can be organized in any way that makes inheritance easy and practical. For example, in the Gemini data reduction package, `geminidr`, along with the instrument package, there is a `gemini` package for software that can apply to all or multiple instruments, and a `core` package for primitives that are generic, like for stacking or photometry. In the Recipe System, the default data reduction package is `geminidr`. This can be changed by setting the value of the `drpkg` attribute to the `Reduce` class or with the `--drpkg` options on the `reduce` command line.

Mode

Programmatically, the mode is a label – a string literal – by which recipe libraries are differentiated. The modes are represented in the data reduction package as subdirectory to the `recipes` directory in an "instrument package". For example, mode `sq` is associated with the directory `recipes/sq`. Only the `sq` recipes will be found there. In principle, the mode names should indicate or hint at the recipes' purpose or to the quality of the products.

The mode is an attribute of the `Reduce` class and can be set on the `reduce` command line with the `--qa` and `--ql` flags ("quality assessment" and "quicklook", respectively) with the `sq` mode as the default. The mode specified for pipeline processing can be used as a flag in a primitive too, for example if a primitive is to behave differently depending on mode.

Recipes organized by modes can differ for whatever operational reasons you might have. The Gemini quality assessment recipes, mode `qa`, measure sky condition metrics at various stages of the reduction. That is not done in science quality reduction recipes, mode `sq`, but additional care is made to sky subtraction, for example. The Mode is therefore related to the desired product the recipe is to deliver.

Primitive

A DRAGONS primitive is a method of a primitive class, also called a primitive set. Primitive sets can be associated to specific Astrodata tags that the Recipe System can match to data. A primitive is expected to be a meaningful data processing step and named appropriately, for example, `biasCorrect` will apply the bias correction. This is a guideline and the Recipe System has no technical requirements for this.

Primitive Class

The Recipe System matches the data to the most appropriate Primitive Class, also called Primitive Set. The association is made using the Astrodata tags of the input data, and the tagset attached to each primitive class. Each primitive class must have a `tagset` class attribute assigned with a Python set containing the relevant Astrodata tags, eg. `tagset = set(["GEMINI", "GMOS"])`. The class that matches the greatest number of tags wins the contest and gets selected.

The primitive classes can make use of inheritance and mix-ins to collect a complete set of primitives applicable to the data being reduced.

Recipe

A recipe is a python function defined for specific instruments, instrument modes, and pipeline modes (see above, `mode`). A recipe function receives one parameter: an instance of a primitive class, also called a "primitive set". The recipe can then use any primitives from that set.

The recipes are stored in a Recipe Library, a Python module, see below.

Recipe Library

A Recipe Library is a Python module that stores recipes. The Recipe Library is given a tagset for data to recipe mapping by the Recipe System. All the recipes in a Recipe Library must therefore apply to the same Astrodata tags. The Astrodata tagset is stored in the module variable `recipe_tags`.

Each library must have a recipe assigned as "default". The module variable `_default` is set to the name of the default recipe.

The Recipe System finds the Recipe Library that best matches the data based on tagset and mode. Then it picks from it the default recipe, or the user-specified recipe from that library. The primitive set is selected and passed to the selected recipe, completing the mapping.

Tagset

A Tagset is a Python set of Astrodata tags (see *Astrodata documentation*). A dataset opened with Astrodata will be recognized and assigned a set of *tags*. These tags are used by the Recipe System map the data to the most appropriate recipe library and the most appropriate primitive set.

A recipe library announces the tags it applies to with a tagset stored in the module `recipe_tags` variable. A primitive class uses the class attributr `tagset` to store the applicable tags. The Recipe System maps all the tagsets together to find the best mapping solution.

For example, a datasets is assigned the following tags:

```
>>> ad = astrodata.open('N20170609S0160.fits')
>>> ad.tags
set(['RAW','GMOS','GEMINI','NORTH','SIDEREAL','UNPREPARED','IMAGE','ACQUISITION'])
```

The Recipe System will match that data to the recipe library and primitives with the following tags:

```
Recipe Library with: recipe_tags = set(['GMOS', 'IMAGE'])
Primitive Class with : tagset = set(["GEMINI", "GMOS", "IMAGE"])
```

Tagset matching by the Mapper classes are discussed in greater detail in subsequent chapters of this document, *Chapter 3, The Mappers*, and *Chapter 4, Using The Mappers API*.

2.2 Outline of the Recipe System

In this section we provide a functional overview of the Recipe System, and describe in more detail some of the key components of the complete reduction ecosystem.

2.2.1 Functional overview

The complete reduction ecosystem is represented in *Figure 2.1* with emphasis on how the Recipe System automates and supports the data reduction. It illustrates the system's relationship to instrument packages and the calibration manager.

Fig. 1: Figure 2.1: Schematic Diagram of Recipe System Components and the supporting Calibration Request Service

Let us go through that diagram.

1. The command line interface, `reduce`, provides users command access and execution from the terminal. (*Reduce and Recipe System User Manual.*)
2. The `Reduce` class receives input datasets and parameters either from `reduce` or directly through the `Reduce` API (*Reduce and Recipe System User Manual.*). `Reduce` parses the input arguments and opens the input datasets. When run, it will send the first input to the Mappers.
3. The *Mappers*, both RecipeMapper and PrimitiveMapper, conduct best matching tests on recipe libraries and primitive classes and return the best matched objects. The Astrodata tags and the recipe libraries and primitive classes tagsets are used to do the match.

4. The instrument data reduction (DR) package is a collection of modules that provide data reduction classes (primitives) and recipe libraries, and any supporting software like lookup tables. The instrument DR packages are not part of the Recipe System, they are add-ons specific to the instruments being supported. The Recipe System probes and searches those packages for matching primitive sets and recipes. In DRAGONS, the instrument DR packages, and some generic primitive packages, are found under `geminidr`.
5. The Calibration Request Service provides a functional interface between primitives requesting calibration files (biases, flats, etc.) and the designated calibration manager.
6. The Calibration Manager is an independent component, not part of DRAGONS, that contains the calibration association rules and interacts with a database that stores the processed calibrations. It accepts calibration requests passed by the Calibration Request Service at the behest of the primitive calls. The Calibration Manager can be *Local Calibration Manager* distributed as `GeminiCalMgr` for use by individuals, or the Gemini internal facility calibration manager. The latter is for Gemini Operations needs only (for reference, Fitsstore). In either case, the data's metadata are used, along with a set of rules, to determine a best available match for the requested calibration type and return a full path name (local) or a URL (internal Gemini manager) to the file.

It is worth noting that all components discussed here operate and communicate using the common grammar provided by the `astrodata` data abstraction.

2.2.2 `reduce` and `Reduce`

`reduce` is the easiest way to invoke the Recipe System. It passes command line options to the `Reduce` class, which then invokes the mappers. Those, in turn, use arguments to locate and identify the best applicable primitive classes and recipes. For most users, `reduce` will be the common way to process datasets with the Recipe System.

The `Reduce` class can be used directly for a programmatic invocation of the reduction rather than using the terminal.

Usage of both `reduce` and `Reduce` is documented in the *Reduce and Recipe System User Manual*.

The `reduce` script itself is really light weight and mostly just a wrapper around `Reduce`. It sets up a logger and then uses the same parser that `Reduce` also has access to, `buildParser`. Then it is off to `Reduce` to to run the show.

A `Reduce` instance can be created with or without arguments. The argument is a string representing the command line of `reduce`. When that argument is provided, `Reduce` will call `buildParser` on it. The instance attributes can also be set individually. When using the API, a logger must be set ahead of time, `Reduce` will not create one, yet it expects to be able to write to one. The main public method of `Reduce` is `runr()` which is responsible for applying the mapper-returned primitive instance to the mapper-returned recipe function, at which point, processing begins. Note that `runr` has logic to recognize the name of a primitive and to run that specific primitive rather than a recipe. Of course, the primitive will be coming from a tagset matching primitive set.

2.2.3 Mappers

The mapper classes are the core of the Recipe System and provide the means by which the Recipe System matches input datasets to processing routines. When applicable primitive classes and recipes are found, the mappers return objects of the appropriate kind to the caller; the `PrimitiveMapper` returns an instance of the applicable primitive class; the `RecipeMapper` returns the actual recipe function object from the applicable recipe library.

There are two functional mapper classes, `RecipeMapper` and `PrimitiveMapper`, which are subclassed on the base class, `Mapper` . These classes and their modules are located in `recipe_system.mappers`.

Mappers are discussed more fully in the *next chapter*.

2.2.4 Instrument Data Reduction Packages

The data reduction packages are not components of the Recipe System. They stand on their own. They provide the means, or instructions, for reducing data and, therefore, at least one such package is required for the Recipe System to function.

The data reduction packages provide some hooks that the Recipe System depends on to map recipes and primitives to the data.

Instructions on how to structure a data reduction package for use by the Recipe System are provided in *appendix*.

The primitive class signature must be able to accept this instantiation call:

```
primitive_actual(self.adinputs, mode=self.mode, ucals=self.usercals,
                 uparms=self.userparams, upload=self.upload)

 adinputs: Python list of AstroData objects
 mode:     One of 'sq', 'qa', or 'ql'
 ucals:    Python dict with format
              {(<data_label>, <type_of_calib>): <calib_filename>},
              one key-value pair for each input, with the type of
              calibration matching one from the list in
              cal_service.transport_request.CALTYPES.
 uparms:   Python dict with format ``{'<prim>:<param>': <value>}``
 upload:   Python list of any combination of 'calibs', 'metrics', or
              'science'.
```

In `geminidr`, the primitive classes use a decorator to process those inputs.

The recipes must be located in subdirectory named after the *mode*. For example:

```
<inst_pkg>/
    __init__.py
    recipes/
           __init__.py
           qa/
           sq/
           .../
```

While it is entirely possible to allow unrestricted naming of subpackages and modules within an instrument data reduction package, the Recipe System is optimized to search packages of a certain form. In particular, some optimization allows the mapping algorithms to bypass code defined in the `lookups/` directory where Gemini puts static inputs like look-up tables and bad pixel masks. Because the Recipe System conducts depth-first searches, the optimization expedites mapping by known exclusion: bypassing subpackages and modules that are known not contain primitives or recipes.

Refer to the *appendix* for more a more complete discussion.

2.2.5 Calibration Request Service

The Calibration Request Service provides a functional interface to a local calibration manager (GeminiCalMgr) or the Gemini Observatory facility calibration manager (fitsstore) The Calibration Request Service does **not** communicate with the Gemini Observatory Archive.

Primitives requiring **processed** calibration files (biases, flats, etc.) will use this functional interface to make calibration requests. These requests are served by the calibration manager in real time. This is a *JIT* (just in time) service. (See the *Appendix* for more information about why *JIT* calibration service is necessary.)

Calibration requests are built from the Astrodata descriptors and tags, and the requested calibration type (flat, dark, etc). The calibration request is processed by the calibration manager's association rules to find the best match.

The details of the request depends on the calibration manager being used. That is set upon import of the `calrequestlib` package. The `calibration_search` module variable is set via the *cal_search_factory()* function to either the `calibration_search` method in the `LocalManager` class or the `calibration_search` function in the `transport_request` module. The former applies when the local calibration manager is used, the latter when the Gemini internal `fitsstore` server is used.

In the case of the local calibration manager, the manager's `get_cal_object` function is accessed directly. In the case of the internal fitsstore server, an HTTP POST request is made on the server.

In both cases, the return value is a tuple with the URLs to the processed calibrations and the correspond md5 sums.

The Calibration Request Service is responsible for determining whether the matched calibration has already been downloaded from `fitsstore` and cached by verifying the md5 sums. If the file is in the cache, the path to the local file is returned rather than fetching the file again. If the file has not been cached, then the request service downloads the file using the returned URL and stores it locally, then that newly downloaded file is passed to the calling primitive. The storage directory is called `calibrations` in the root directory of the tool making the request.

In the case of the local calibration manager, the data are already local. The calibration manager only stores filename and path, not the data. The path returned is the path to the local version that was added by the user to the database.

2.2.6 Calibration Manager

The Calibration Manager is an external component to the Recipe System and even DRAGONS itself. The Recipe System currently uses two types of calibration manager.

The original calibration manager is one used internally at Gemini. It is associated with a large database that stores the data too. For external users, a light weight local calibration manager is available instead.

The local calibration manager uses a sqlite database to store the location information of the calibrations processed by the user. Since the data were processed locally, there is no need to store the data, just the name and the path to the data.

What both calibration managers share are the calibration associations rules, rules that will identify the best processed calibrations for a given Gemini observation. Those rules are the same as the rules used by the Gemini Observatory Archive. The internal database is in fact using exactly the same software as the GOA. The local calibration manager uses a subset of the code plus a couple extra routines.

The Recipe System knows how to make requests to either of those two sources of processed calibration. For the local calibration manager, the Recipe System provides the `caldb` facility to create and populate (or de-populate) the local database.

The internal Gemini data manager is obviously very Gemini-centric, by necessity. The local calibration manager, distributed as GeminiCalMgr, is also, unfortunately still quite Gemini-centric. The ORMs are designed for Gemini data. It might be possible for a third-party to replace the ORMs and the calibration rules to match their data's needs.

CHAPTER 3

The Mappers

The mappers are the core of the Recipe System. The mappers match recipes and primitives to the data being reduced.

Normally called from the `Reduce` class (which might have been called by the command line `reduce`), the mappers will search a data reduction package (DR package) specified by the argument `drpkg` for recipes and primitives matching the AstroData tags of the data to be processed. (The default `drpkg` is DRAGONS `geminidr` package.)

The search for matching primitives and recipes is optimized for a certain directory structure. Other structures would work though. In the *appendix*, we introduce a tool to easily create the data reduction package structure that will be most efficient. The reader can also look at `geminidr` for a working example.

The mapping algorithm uses attributes of an `AstroData` instance to first determines the applicable instrument package defined under the DR package, for example `gmos` in `geminidr`. Then the algorithm conducts a search of that instrument package looking for specific attributes: a class attribute called `tagset` defined in discoverable primitive classes, and a module attribute, called `recipe_tags`, defined in recipe library modules. The mappers look for the "best" match between an AstroData object's tags and the tagset defined in recipe libraries and primitive classes. The best match requires that the primitive and recipe tagsets are a *complete subset* of the larger set of tags defined for the input dataset. The association with the greatest number of matched tags wins the test.

3.1 Mapper Class

`Mapper` serves as the base class for the other Recipe System mapper classes. The base `Mapper` class defines *only* the initialisation function. Subclasses inherit and do not override `Mapper.__init__()`.

It is important to note that only the first AstroData dataset in the list of input datasets is used to set the tags and the import path of the instrument package. It is assumed that all the datasets in the list are of the same type and will be reduced with the same recipes and primitives.

Class Mapper *(adinputs, mode='sq', drpkg='geminidr', recipename='_default',

usercals=None, uparms=None, upload=None)*

Arguments to __init__

adinputs A `list` of input AstroData objects (required).

mode A `str` indicating mode name. This defines which recipe set to use. Default is `'sq'`. See the *mode definition* for addtional information.

drpkg A `str` indicating the name of the data reduction package to inspect. The default is `geminidr`. The package *must* be importable and should provide instrument packages of the same form as defined in *appendix*.

recipename A `str` indicating the recipe to use for processing. This will override the mapping in part or in whole. The default is "_default". A recipe library should have a module attribute `_default` that is assigned the name of the default function (recipe) to run if that library is selected. This guarantees that if a library is selected there will always be a recipe matching `recipename` to run.

`recipename` can also be the name of a specific recipe that will be expected to be found in the selected recipe library.

To completely bypass the recipe mapping and use a user-provided recipe library and a recipe within, `recipename` is set to

<(path)library_file_name>.<name_of_recipe_function>.

Finally, `recipename` can be set to be the name of a single primitive. In that case, the primitive of that name in the mapped primitive set will be run.

usercals A `dict` of user-specified calibration files, keyed on calibration type. Calibration types are set from the calibration manager. There is a copy of the list of acceptable types in `transport_request.py`. E.g., `{'processed_bias':'foo_bias.fits'}`

uparms A `list` of tuples representing user parameters passed via command line or the `Reduce` API. Each may have a specified primitive. If no primitive is specified, the value is applied to all parameters matching the name in any primitives.

E.g., [('param','value'), ('<primitive_name>:param1','val1')]

upload *Internal to Gemini only.* Default is None.

A `list` of strings indicating the processing products to be uploaded to the internal Gemini database. Allowed values are `metrics`, `calib`, `science`. The latter is not implemented. For example, the night time quality assessment pipeline produces sky quality metrics. When `upload = ['metrics']`, the `geminidr` primitives `measureBG`, `measureCC`, and `measureIQ` will upload their results to the internal database.

Attributes

adinputs See above.

mode See above.

pkg A `str` set from the `instrument` descriptor of the first AstroData object in the `adinputs` list. Uses the generic name (e.g. gmos rather than gmos-s) and lowercase. The `pkg` string must match the name of an instrument package. ??KL, add reference to appendix This is an optimization feature that will limit the primitive and recipe search to the instrument package that matches the input data.

dotpackage A `str` set from the value of `drpkg` and `pkg` to represent the dot-path to the package. Eg. `gemini_instrument.gmos`. This will be use for dynamic imports.

recipename See above.

tags A Python `set`, set from the `tags` of the first AstroData object in the `adinputs` list.

usercals The value of `usercals` as passed to `__init__` or an empty dictionary if nothing is passed.

userparams A `dict` version of list of tuples found in `uparms`. The dictionary version can be used more directly by the primitives.

_upload, property upload *Internal to Gemini only.* Default is None.

A `list` of `str` to identify the types of upload that are to be performed. The valid strings are: "metrics", "calib", "science". The `__init__` uses a property to parse and this attribute.

3.2 PrimitiveMapper

`PrimitiveMapper` is subclassed on `Mapper` and does *not* override `__init__()`. `PrimitiveMapper` implements the primitive search algorithm and provides one public method on the class: `get_applicable_primitives()`.

Class PrimitiveMapper *(adinputs, mode='sq', drpkg='geminidr', recipename='_default',

usercals=None, uparms=None, upload=None)*

Arguments to __init__ See *Mapper* above.

Attributes See *Mapper* above.

Public Methods

get_applicable_primitives *(self)* Search the data reduction and instrument packages for the best matching primitive class for the input dataset.

Returns Instance of the selected primitive class.

The primitive set search is conducted by comparing the AstroData `tags` attribute of the first input dataset to the `tagset` attribute of each primitive class. The best matched primitive set should, in principle, be found in an instrument package that matches the instrument descriptor of the input datasets. In fact, the way the search is optimized right now, it enforces that principle.

As the search of instrument primitive classes progresses, modules are inspected, looking for class objects with a *tagset* attribute. A tagset match is assessed against all previous matches and the best matching class is retrieved and instantiated with all the appropriate arguments.

A match requires that the primitive class *tagset* be a subset of the AstroData *tags* descriptor. The *best match* is the one with the largest *tagset*. If two or more primitive classes return best matches with the same number of tags in their *tagset*, then the current algorithm will only return the first *best match* primitive class it has encountered. It is therefore very important to be specific with the *tagset* attributes to avoid such multiple match situation and to ensure only one true best match.

The `get_applicable_primitives()` method returns this instance of the best match primitive class. The object returned will be the actual instance and usable as such as an argument to a recipe function. The list of AstroData objects given as input to PrimitiveMapper is used to instantiate the chosen primitive class.

The instantiation of the primitive class by `get_applicable_primitives()` implies an API requirement on the Primitive class. It must be possible to intantiate a primitive class with the following call:

```
PrimitiveClassName(self.adinputs, mode=self.mode, ucals=self.usercals,
                   uparms=self.userparams, upload=self.upload)
```

where `self` is an instantiated PrimitiveMapper.

3.3 RecipeMapper

`RecipeMapper` is subclassed on `Mapper` and does *not* override `__init__()`. `RecipeMapper` implements the recipe search algorithm and provides one public method on the class: `get_applicable_recipe()`.

Class RecipeMapper *(adinputs,mode='sq',drpkg='geminidr',recipename='_default',

usercals=None, uparms=None, upload=None)*

Arguments to __init__ See *Mapper* above.

Attributes See *Mapper* above.

Public Methods

get_applicable_recipe ***(self)*** Search the data reduction and instrument packages for the best matching recipe library (module) for the input dataset. Then returns either the *default* recipe or the one named in `recipename`.

Returns A function defined in an instrument package recipe library.

The recipe library search is conducted by comparing the AstroData `tags` attribute of the first input dataset to the `recipe_tags` module attribute of each recipe library in the `mode` subdirectory of the instrument package. The best matched recipe library should in principle be found in an instrument package that matches the instrument descriptor of the input datasets. In fact, the way the search is optimized right now, it enforces that principle.

The `mode` narrows the recipe search in the instrument package to the corresponding subdirectory, while the AstroData tags are used to locate the desired recipe library within that `mode` subdirectory. As the search of instrument recipe modules (libraries) progresses, modules are inspected, looking for a `recipe_tags` attribute. A recipe tags match is assessed against all previous matches and the best matching recipe library is imported. The *default* or the named recipe function is retrieved from the recipe library.

A match requires that the primitive class *tagset* be a subset of the AstroData *tags* descriptor. The *best match* is the one with the largest matching subset of the data's tags attribute. If two or more primitive classes return best matches with the same number of tags in their *tagset*, then the current algorithm will only return the first *best match* primitive class it has encountered. It is therefore very important to be specific with the *tagset* attributes to avoid such multiple match situation and to ensure only one true best match.

The `get_applicable_recipe()` method returns this *best match* recipe function to the caller. This is a function object and is callable.

3.4 The Handling of `recipename`

The input argument `recipename` is multiplexed. The default value is the string *default* that matches a module attribute in the recipe library identifying which of the recipes is to be considered the default.

`recipename` may be set to the name of a specific recipe (function) within a recipe library. This will override the default setting. Of course, the function must be present in the *best match* recipe library.

The `recipename` can also specify a user-provided recipe. The syntax for this form is `<recipe_library>.<recipe>`. If the <recipe_library> string does not contain the full path to the module, the current directory is assumed to be the location of the library. The `RecipeMapper` first tries to find such a recipe. If it is not found, then the mapper begins the process of searching for the *best match* recipe in the data reduction package.

Finally, the `recipename` can be the name of a primitive rather than the name of a recipe (eg. `reduce N20120212S0012.fits -r display`). In that case, the `RecipeMapper` will fail to find a matching recipe. Recognition of the string as a valid primitive is done in the `runr()` method of `Reduce`. The primitive must be one of the primitives from the *best match* primitive set.

CHAPTER 4

Using the Mappers API

The `Mapper` base class provides no functionality, rather its purpose is to define all the attributes for instances built from subclasses of Mapper. Though not strictly an abstract class, the `Mapper` base class cannot be used on its own.

The subclasses, `PrimitiveMapper` and `RecipeMapper`, do not override `Mapper.__init__()`. They implement mapping routines aiming to find the primitives and recipes that best match the input data.

The mappers require a Data Reduction Package itself containing Instrument Packages. We discuss these packages and how to build the basic directory structure for them in the *Appendix*.

Below, we use the `geminidr` data reduction package when we need an example. Note that `geminidr` is the default package unless it is overridden with the `drpkg` argument.

4.1 Selecting Primitives with PrimitiveMapper

Primitive classes are defined in a Data Reduction (DR) Package. For the Gemini instruments that is the `geminidr` package. The DR Package to use by the mapper is set by the `drpkg` argument to mapper.

These primitive classes define methods, referred to as *primitives*, that provide data processing functionality. Primitive classes in a DR Package are structured hierarchically and may employ multiple inheritance.

In Figure 4.1 below, we show an example of primitive class inheritance as done in `geminidr`, with `GMOSImage` the primitive class inheriting and mixing-in the other classes. In general, the class inheriting others will also have longer, more specific *tagsets*. This will ensure that the mapping for a GMOS image, if we follow the example in Figure 4.1, does return the *GMOSImage* class as the best match, not *GMOS*, not *GEMINI*.

The GMOSImage primitive class, by inheriting all of the classes above it and hence all their methods (aka primitives) becomes a complete set of primitives that can be applied to a GMOS IMAGE. It becomes a *primitive set*.

Generic primitive classes like CCD, Image, Photometry in Figure 4.1 normally should have empty *tagsets*.

Note: Hereafter, a Primitive class may be referred to as a "primitive set" or just "primitives". Through inheritance, a primitive class collects many primitives from higher in the hierarchy. The primitive set implies the whole collection more clearly than just referring to the primitive class.

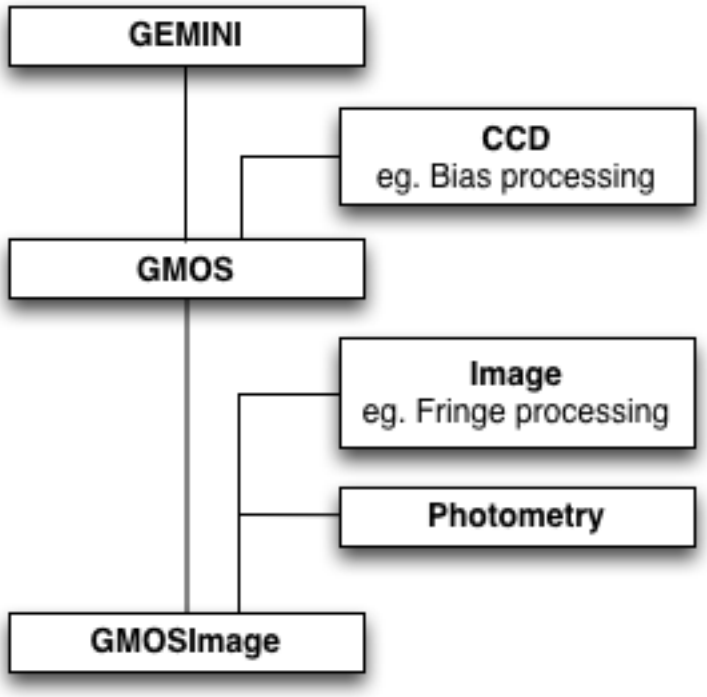

Fig. 1: Figure 4.1: An example of primitive class inheritance. (A section of `geminidr`.)

4.1.1 Mapping Data to Primitives

At a minimum, `PrimitiveMapper` requires as input a list of AstroData objects. Only the first object in the list will be used for mapping. The whole list will be passed to the mapped primitive set once found and instantiated.

The mapping is done by matching the primitive set's `tagset` attribute to the AstroData tags of the first input element.

The primitive set is obtained as follow, where `ad` is an AstroData object:

```
>>> from recipe_system.mappers.primitiveMapper import PrimitiveMapper
>>> tags = ad.tags
>>> instpkg = ad.instrument(generic=True).lower()
>>> pmapper = PrimitiveMapper(tags, instpkg)
>>> pclass = pmapper.get_applicable_primitives()
>>> p = pclass([ad])
```

This `p` can now be passed to the best match recipe.

Example

Let us discuss an example. Let us use a raw GMOS image.

```
 >>> ad = astrodata.open('S20161025S0111.fits')
 >>> ad.tags
set(['RAW', 'GMOS', 'GEMINI', 'SIDEREAL', 'UNPREPARED', 'IMAGE', 'SOUTH'])
```

The `PrimitiveMapper` uses these tags to search the `geminidr` packages, or the data reduction package specified in the mapper attribute `drpkg`. The first step in the search is to narrow it to the applicable instrument package, in

this case, `gmos`. The mapper enforces the rule that the directory name for the instrument package be the lower case version of the AstroData descriptor `instrument()`. This speeds up the search but it is also a limitation.

The `PrimitiveMapper` instance stores the name of the package it will search in the `dotpackage` attribute. For example, using the GMOS `ad`:

```
>>> from recipe_system.mappers.primitiveMapper import PrimitiveMapper
 >>> pmapper = PrimitiveMapper([ad])
 >>> pmappmer.dotpackage
'geminidr.gmos'
```

The public method `get_applicable_primitives()`, when invoked, launches the search for the most appropriate primitive class. The search itself is focused on finding class objects with a `tagset` attribute.

```
>>> pclass = pmapper.get_applicable_primitives()
```

In the case of GMOS, the `tagsets` in primitive classes hierarchy (Figure 4.1 above) are:

```
class Gemini( ...  ):
   tagset = set(["GEMINI"])

class GMOS(Gemini, ... ):
   tagset = set(["GEMINI", "GMOS"])

class GMOSImage(GMOS, ... ):
   tagset = set(["GEMINI", "GMOS", "IMAGE"])
```

The function looks to match the `tagset` of the primitive class it finds to the *AstroData tags*. A successful match requires `tagset` to be a subset of the Astrodata tags. The match with the largest subset wins the contest. In our example, the class `GMOSImage` wins. Indeed:

```
>>> pclass
<class 'geminidr.gmos.primitives_gmos_image.GMOSImage'>
```

To use this selection of primitives, one then needs to instantiate it:

```
>>> p = pclass([ad])
```

This can be used by a recipe.

4.2 Selecting Recipes with RecipeMapper

Recipes are functions that receive a single argument: an instance of a primitive class. Recipe functions are not classes and do not (cannot) inherit. The recipe simply defines the primitives to be used and the order in which the primitive calls are made on the data.

Recipe functions are defined in python modules which we refer to as recipe libraries. The location of those modules in a data reduction package must obey some rules; they must be located in an instrument package under a subdirectory named `recipes` and therein in a subdirectory that matches the `mode` setting. The `RecipeMapper` searches only such directories. For example:

```
my_drpkg/instrumentX/recipes/modeY/recipes_IMAGING.py
```

4.2.1 Mapping Data to Recipes

At a minimum, `RecipeMapper` requires as input a list of AstroData objects. Only the first object in the list will be used for mapping. The mapping is done by matching the recipe library's `recipe_tags` module attribute to the AstroData tags of the first input element.

The recipe is obtained as follow, where `ad` is an AstroData object:

```
>>> from recipe_system.mappers.recipeMapper import RecipeMapper
>>> tags = ad.tags
>>> instpkg = ad.instrument(generic=True).lower()
>>> rmapper = RecipeMapper(tags, instpkg)
>>> recipe = rmapper.get_applicable_recipe()
>>> recipe.__name__
reduce
```

The recipe can then be run by passing it a primitive set (see the section above on primitive mapping):

```
>>> recipe(p)
```

Example

Let us discuss an example. Let us use a raw GMOS image.

```
 >>> ad = astrodata.open('S20161025S0111.fits')
 >>> ad.tags
set(['RAW', 'GMOS', 'GEMINI', 'SIDEREAL', 'UNPREPARED', 'IMAGE', 'SOUTH'])
```

The `RecipeMapper` uses these tags to search the `geminidr` packages, or the data reduction package specified in the mapper attribute `drpkg`. The first step in the search is to narrow it to the applicable instrument package, in this case, `gmos`. The mapper enforces the rule that the directory name for the instrument package be the lower case version of the AstroData descriptor `instrument()`. This speeds up the search but it is also a limitation.

Next, it narrows the search to the subdirectory `recipes` and then the subdirectory therein named after the `mode`. The default `mode` in `RecipeMapper` is `sq`. In our example, this means `gmos/recipes/sq/`

The `RecipeMapper` instance stores the name of the instrument, and thus the name of the instrument package in the `pkg` attribute, which is set upon instantiation from the `.instrument()` descriptor of the first AstroData object in the input list. For example, using the GMOS `ad`:

```
>>> from recipe_system.mappers.recipeMapper import RecipeMapper
>>> tags = ad.tags
>>> instpkg = ad.instrument(generic=True).lower()
>>> rmapper = RecipeMapper(tags, instpkg)
>>> recipe = rmapper.get_applicable_recipe()
```

The public method `get_applicable_recipe()`, when invoked, launches the search for the most appropriate recipe library. The search itself is focused on finding modules with a `recipe_tags` attribute.

```
>>> recipe = rmapper.get_applicable_recipe()
```

The function looks to match the `recipe_tags` of the recipe libraries it finds to the *AstroData tags*. A successful match requires `recipe_tags` to be a subset of the Astrodata tags. The match with the largest subset wins the contest. In our example, the recipe library `recipes_IMAGE` of mode `sq` of GMOS wins. In there, the default recipe is named `reduce`.

```
>>> recipe.__name__
reduce
>>> recipe.__module__
'geminidr.gmos.recipes.sq.recipes_IMAGE'
```

4.2.2 Selecting External (User) Recipes

It is possible to bypass the recipe mapping entirely and request the use of a user recipe library and recipe. It is simply a matter of setting the `recipename` to the path to the recipe library and to the name of the recipe when the `RecipeMapper` is instantiated. The `RecipeMapper` imports the file and returns the recipe function object.

While some users may have set their `PYTHONPATH` to include such arbitrary locations, which would allow the `myrecipes` module to be imported directly, most people will not have such paths in their `PYTHONPATH`, and would not be able to directly import their recipe file without modifying their environment. Using the RecipeMapper lets users avoid this hassle because it handles import transparently.

```
>>> rm = RecipeMapper(adinputs, recipename='/path/to/myrecipes.myreduce')
>>> recipefn = rm.get_applicable_recipe()
>>> recipefn.__name__
'myreduce'
```

Note that for user supplied recipe libraries and functions, the `mode`, the `pkg`, the `drpkg`, and the tags are irrelevant. Essentially, passing a user-defined recipe to the RecipeMapper tells the mapper, "do not search but use this." In these cases, it is incumbent upon the users and developers to ensure that the external recipes specified are actually applicable to the datasets being processed.

4.2.3 Primitives and Recipes, Together at Last

To summarize the mapper usage described in this chapter, to launch a reduction one does:

```
>>> tags = ad.tags
>>> instpkg = ad.instrument(generic=True).lower()
>>> rmapper = RecipeMapper(tags, instpkg)
>>> pmapper = PrimitiveMapper(tags, instpkg)
>>> recipe = rmapper.get_applicable_recipe()
>>> pclass = pmapper.get_applicable_primitives()
>>> p = pclass([ad])
>>> recipe(p)
```

This is essentially what the `Reduce` class does.

4.3 Using the Primitives Directly

The primitves in the selected primitive set can be used directly. The recipe is just a function that calls some primitives from the set in some pre-defined order. This usage can be useful for testing, debugging or interactive use.

The primitive set is obtained as shown in the previous section, using the `PrimitiveMapper` and its `get_applicable_primitives` method. To retrieve the list of primitives from the primitive set, one can do this:

```
>>> import inspect
>>> for item in dir(p):
...     if not item.startswith('_') and \
...         inspect.ismethod(getattr(p, item)):
...         print(item)
...
```

The primitive set has been initialized with a list of AstroData objects. Running a primitive on them only requires calling the primitive as a method of the primitive set.

The exact call depends on the primitive itself. In `geminidr`, the primitives use a `streams` attribute to store the AstroData objects being processed in such a way that those objects do not need to be passed to the primitives when called. For example:

```
>>> p.prepare()
>>> p.addVAR()
```

The syntax above uses *streams* to pass the data along from one primitive to the next. The streams are attributes of the `geminidr PrimitiveBASE` class that all `geminidr` primitive classes inherit. When writing a new data reduction package, the use of that base class, and the `parameter_override` decorator in `recipe_system.utils.decorators`, is required to benefit from the *streams* system.

It is also recommended, and indeed the primitives in `geminidr` do that, to have the primitives methods return the modified output AstroData objects. When *streams* are used, return values are not necessary, but for debugging, testing, or exploration purposes it can be handy. For example, one can do this:

```
>>> intermediate_outputs = p.prepare()
```

A logger is currently required. The logger in `gempy.utils` called `logutils` is used by the recipe_system. The output will go to both stdout and a logfile. If the logfile is not defined, it leads double-printing of the logging on stdout. To avoid the double- printing the logfile name must be set to something. If you do not want to write a logfile to disk, on Unix systems you can set the file name to `/dev/null`.

```
>>> from gempy.utils import logutils
>>> logutils.config(file_name='/dev/null')
```

Here is what the stdout logging looks like when a primitive is run directly:

```
>>> p.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 GMOS
         .
```

(continues on next page)

(continued from previous page)

```
    .
  .
  [<gemini_instruments.gmos.adclass.AstroDataGmos object at 0x11a12d650>]
```

Of interest when running the primitives in this way are the contents of the `streams` and `params` attributes of the primitive set.

The *streams* and *params* are features of the `geminidr` `PrimitiveBASE` class, the `parameter_override` decorator, and the `pexconfig`-based parameter system used in `geminidr`. For the time being, developers of new DR packages must use those systems.

The main stream that allows the data to be passed from one primitive to the other without the need of arguments is `p.streams['main']`. When the primitive set is instantiated by `get_applicable_primitives`, the input AstroData objects are stored in that stream. If for some reason you need to repopulate that stream, do something like this:

```
>>> new_inputs = [ad1, ad2, ... adn]
>>> p.streams['main'] = new_inputs
```

The input parameters to the primitives are stored in the primitive set in `p.params`. For example, to see the parameters to the `prepare` primitive and their current settings:

```
>>> p.params['prepare'].toDict()
OrderedDict([('suffix', '_prepared'), ('mdf', None), ('attach_mdf', True)])
```

Or, prettier

```
>>> print(*p.param['prepare'].toDict().items(), sep='\n')
('suffix', '_prepared')
('mdf', None)
('attach_mdf', True)
```

If you need guidance as to the recommended sequence of primitives, you can inspect the recipes returned by the `RecipeMapper`.

```
>>> from recipe_system.mappers.recipeMapper import RecipeMapper
>>> tags = ad.tags
>>> instpkg = ad.instrument(generic=True).lower()
>>> rmapper = RecipeMapper(tags, instpkg)
>>> recipe = rmapper.get_applicable_recipe()
>>> recipe.__name__
'reduce'
>>> import inspect
>>> print(inspect.getsource(recipe.__code__))
def reduce(p):
    """
    This recipe performs the standardization and corrections needed to
    convert the raw input science images into a stacked image.

    Parameters
    ----------
    p : PrimitivesBASE object
        A primitive set matching the recipe_tags.
    """

    p.prepare()
```

(continues on next page)

(continued from previous page)

```
    p.addDQ()
    p.addVAR(read_noise=True)
    p.overscanCorrect()
    p.getProcessedBias()
    p.biasCorrect()
    p.ADUToElectrons()
    p.addVAR(poisson_noise=True)
    p.getProcessedFlat()
    p.flatCorrect()
    p.getProcessedFringe()
    p.fringeCorrect()
    p.mosaicDetectors()
    p.detectSources()
    p.adjustWCSToReference()
    p.resampleToCommonFrame()
    p.flagCosmicRaysByStacking()
    p.scaleByExposureTime()
    p.stackFrames(zero=True)
    p.storeProcessedScience()
    return
```

CHAPTER 5

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

Data Reduction Packages

In the context of the Recipe System, a *Data Reduction Package* is a python package containing one or more subpackages, each providing instrument-specific sets of data processing primitive classes and recipe libraries. These packages provide attributes on certain components of the package, which make them discoverable by the Recipe System. Developers are entirely free to build their own data reduction package, or "dr-package."

As stated at *the beginning of Chapter 4*, the default and only data reduction package provided by DRAGONS is `geminidr`. This package is included in the DRAGONS distribution. Unless specifed otherwise, it is the `geminidr` package that serves targets for the Recipe System mappers. Readers are encouraged to examine the `geminidr` package to familiarize themselves with components.

A.1 Building a new Data Reduction Package

The Recipe System mappers require the primitives and the recipes to be organized under a specific directory structure. To help build that structure the Recipe System provides a script called `makedrpkg` that we will introduce later in this section. First, we address the requirements.

- The Data Reduction (DR) package **must** be importable. (It must have a `__init__.py`.)
- The DR package **must** be found in one of the `sys.path` directories.
- The instrument packages must be found at the first subdirectory level in the DR package.
- The instrument packages (directory name) must be named after the instrument they associate with. The instrument package name **must** match the lower case version of the AstroData descriptor `instrument` for the data it supports.
- The recipes must be in a subdirectory of the instrument package. That directory **must** be named `recipes`. That name is hardcoded in the moduleattribute `RECIPEMARKER` in `utils.mapper_utils`.
- The recipes **must** be assigned a `mode` (one of "sq", "qa", "ql").
- The mode-specific recipes **must** be located in a subdirectory of `recipes`. That directory **must** be named to match the mode.
- The `recipes` directory and the *mode* directories but all have an `__init__.py` in them and be importable.

The directory structure can be created by hand but to simplify the process and avoid mistakes, it is recommended to use the `makedrpkg` script provided with the Recipe System. The script is used from a normal terminal, not from Python. Here is a few usage examples.

Get help:

```
% makedrpkg -h
```

Create `mydrpkg` DR package with a `coolinstrument` instrument package and a `sq` mode subdirectory:

```
% makedrpkg mydrpkg -i coolinstrument -m sq
```

Same as above but with both `sq` and `qa` modes. Note that if a directory already exists it will just be skipped.

```
% makedrpkg mydrpkg -i coolinstrument -m sq qa
```

Add two instrument packages with `qa` mode:

```
% makedrpkg mydrpkg -i instA instB -m qa
```

Add a `sq` mode to an existing instrument package:

```
% makedrpkg mydrpkg -i instA -m sq
```

Once you have that structure in place, the primitives and the parameters modules go in the instrument package main directory, and the recipes in the `recipes/<mode>` directory.

A.2 Using a third-party Data Reduction Package

To activate a specific DR package, the `drpkg` attribute or option (depending on what is being used) needs to be set. The default setting is `geminidr`.

A.2.1 From the `reduce` command line tool

From the `reduce` command line tool one uses the `--drpkg` option. For example:

```
% reduce *.fits --drpkg mydrpkg
```

A.2.2 From the `Reduce` class API

When using the `Reduce` class API, the attribute to set is `drpkg`. For example:

```
>>> from recipe_system.reduction.coreReduce import Reduce
>>> reduce = Reduce()
>>> reduce.drpkg = 'mydrpkg'
```

A.2.3 From the mappers

When using the mappers directly, again the attribute to set in either mapper, `PrimitiveMapper` or `RecipeMapper`, is `drpkg`. For example:

```
>>> from recipe_system.mappers.primitiveMapper import PrimitiveMapper
>>> from recipe_system.mappers.recipeMapper import RecipeMapper
>>> tags = ad.tags
>>> instpkg = ad.instrument(generic=True).lower()
>>> pmapper = PrimitiveMapper(tags, instpkg, drpkg='mydrpkg')
>>> rmapper = RecipeMapper(tags, instpkg, drpkg='mydrpkg')
```

A.3 Requirements for Primitives and Recipes

Instructions on how to write primitives and recipes is beyond the scope of this manual. However, the Recipe System does impose some requirements on the primitives and the recipes. We review them here.

A.3.1 Requirements on Recipes

- A recipe library must contain a module attribute named `recipe_tags` that contains a Python set of the AstroData tags applicable to the library.
- A recipe library must contain a module attribute named `default` that sets the name of the default recipe for this library. The `default` attribute needs to be set below the recipe function definition for Python to pick it up.
- A recipe signature must accept the primitive set as the first argument with no other "required" arguments. Other arguments must be optional.

A.3.2 Requirements on Primitives

The requirements on the primitives are highly Gemini centric for the moment.

- A primitive class must inherit `PrimitiveBASE` class defined in `geminidr/__init__py`, or bring a copy of it in a third-party DR package.
- A primitive class must be decorated with the `parameter_override` decorator located in `recipe_system.utils.decorators`.
- The Gemini fork of LSST pexconfig package, `gempy.library.config`, must be used to handle input parameters.
- The signature of the `__init__` of primitive class should be:

```
def __init__(self, adinputs, **kwargs):
```

- The signature of a primitive, a method of a primitive classe should be:

```
def primitive_name(self, adinputs=None, **params)
```

- Primitive parameter must be defined in a class matching the name of the primitive followed by "Config". For example:

```
class primitive_nameConfig(any_other_parameter_class_to_inherit):
    param1 = config.Field("Description of param", <type>, <default_value>)
```

- Each primitive class must define a `tagset` attribute containing a `set` of AstroData tags identifying which data this primitive class is best suited for. The tags are string literals. This `tagset` attribute is what the primitive mapper uses to find the most appropriate primitive class.

Here is an example putting most of the above requirements to use:

```
from geminidr.core import Image, Photometry
from .primitives_gmos import GMOS
from . import parameters_gmos_image
from recipe_system.utils.decorators import parameter_override

@parameter_override
class GMOSImage(GMOS, Image, Photometry):

    tagset = set(["GEMINI", "GMOS", "IMAGE"])

    def __init__(self, adinputs, **kwargs):
        super().__init__(adinputs, **kwargs)
        self._param_update(parameters_gmos_image)

    def some_primitive(self, adinputs=None, **params):
        [...]
        return adinputs
```

It is technically possible to decide not to use the `parameter_override` decorator. In that case, there will be no transparent passing of the input AstroData objects, all primitives will have to have their parameter explicitely defined when called, the primitive class will have to have exact signature the `get_applicable_primitives` uses when initializing the class. This is not a mode we have experimented with and there might be additional limitations.

The primitive class signature must be able to accept this instantiation call:

```
primitive_actual(self.adinputs, mode=self.mode, ucals=self.usercals,
                 uparms=self.userparams, upload=self.upload)

 adinputs: Python list of AstroData objects
 mode:     One of 'sq', 'qa', or 'ql'
 ucals:    Python dict with format
              {(<data_label>, <type_of_calib>): <calib_filename>},
              one key-value pair for each input, with the type of
              calibration matching one from the list in
              cal_service.transport_request.CALTYPES.
 uparms:   Python dict with format ``{'<prim>:<param>': <value>}``
 upload:   Python list of any combination of 'calibs', 'metrics', or
              'science'.
```

A.3.3 Requirement to use AstroData

The Recipe System expects to work on AstroData objects. In particular, it requires the `tags` to be defined and the `instrument()` descriptor to be defined.

Therefore, for data from a new, still unsupported instrument, the first step is to write the AstroData configuration layer. In DRAGONS, the AstroData configuration layers are found in the package `gemini_instruments`. The convention is to name the modules with the tags and descriptor `adclass.py`. This is just a convention.

For more information on the AstroData configuration, see the .

APPENDIX B

JIT Calibration Request

It is important to understand that when a calibration request is made, "live" metadata are passed to the calibration manager at the current stage of processing. This kind of operation is called "just in time" (jit), which indicates that one only requests a calibration at the processing stage where and when it is needed.

This is necessary because the correct association of a processed calibration product can actually depend on the processing history of the dataset up to the point where the calibration is needed.

For example, data from a CCD comes with an overscan section. The common reduction steps involve correcting for the signal in the overscan and then trimming that section off. When the processing comes to requesting a processed master bias, that bias must have also been corrected for the overscan signal and match in size with the dataset being reduced, it must also have been trimmed. If the calibration request were to be made on the raw CCD frame, before overscan correction and trimming, that processed bias would not be found, or another one, a mismatched one could be.

This principle would work if for some reason the user decides not to subtract the overscan signal. Then a processed bias still containing the overscan signal would be required. (Note that in DRAGONS, a "processed" calibration is expected to be ready to use, without additional processing.)

Therefore, the Recipe System uses JIT calibration requests. The calibration found will match the data at that point in the recipe where the calibration is needed.

APPENDIX C

Glossary

astrodata Part of the DRAGONS package that defines the abstraction layer for observational datasets. The astrodata abstraction and its associated grammar is used extensively by the Recipe System to effect correct processing.

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 Is a method on an `AstroData` instance. A descriptor represents a high-level metadata name and provides access to essential metadata through a uniform, instrument-agnostic interface to the FITS headers. E.g., `ad.gain()`

DRAGONS 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 `DRAGONS` package comprising gemini specific functional utilities.

primitive A function defined within a data reduction instrument package (a "dr" package) that performs actual work on the passed 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's current astrodata 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 A function defined in a recipe library (module), which defines a sequence of function calls. A recipe is a simple python function that recieves an instance of 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 defined recipes and primitives classes. The Recipe System defines a set of classes that exploit attributes on an astrodata instance of a dataset to locate recipes and primitives appropriate to that dataset.

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

tags [tagset] Represent a data classification. A dataset will be classified by a number of tags that describe both the data and its processing state. For example, a typical unprocessed GMOS image taken at Gemini-South would have the following tagset:

```
set(['RAW', 'GMOS', 'GEMINI', 'SIDEREAL', 'UNPREPARED', 'IMAGE', 'SOUTH'])
```

Instrument packages define *tagsets*, which are sets of string literals that describe and the kind of observational data that the package, primitive, or library has been defined to accommodate and process. As an example:

```
set(['GMOS', 'MOS', 'NODANDSHUFFLE')]
```

Index

A

AstroData, 28
astrodata, 28

D

descriptor, 28
DRAGONS, 28

G

gempy, 28

P

primitive, 28

R

recipe, 28
Recipe System, 28
reduce, 28

T

tags [tagset], 28