Keck Data Reduction Pipelines Framework

Release 1.0

Jul 07, 2020

Contents

This is the documentation for Keck Data Reduction Pipeline Framework package (KDRPF).

KDRPF is a general framework capable of running data reduction pipelines developed specifically for Keck instruments, but it can be used to run any type of data reduction pipelines.

It has been developed by the WMKO Data Reduction Project.

This document shows how to install the package, describes the key concepts behind the framework, and shows the steps necessary to run or develop a pipeline.

CHAPTER 1

Requirements

KDRPF has the following requirements:

- *Astropy*
- pandas

CHAPTER 2

Installation

You can download the code by cloning this repository:

```
git clone https://github.com/Keck-DataReductionPipelines/KeckDRPFramework.git
```

Change directory into the KeckDRPF directory and run:

```
python setup.py install
```

Note that you will download the `develop` branch, which is the default branch. To switch to the `master` branch, use:

```
git checkout -b master --track origin/master
```

CHAPTER 3

Framework design and concepts

3.1 Design and basic concepts

This section describes the core ideas behind the design and implementation of the framework.

3.1.1 Setting up and starting the framework

The `framework` module is the core of the system.

To illustrate a possible way to start up the framework, connect it to the pipeline, and then set it in motion, consider this example:

```
from KCWI_PIPELINE.pipelines.kcwi_pipeline import Kcwi_pipeline

pipeline = Kcwi_pipeline()
framework = Framework(pipeline, 'config.cfg')
framework.start()
```

This block of code performs a full start up of the framework, but doesn't do any processing (see **startup_scripts_**).

These are the operations happening behind the scene:

- an instance of the `BasePipeline` class is created and assigned to the `pipeline` variable
- an instance of the `Framework` class is created, using the pipeline and a configuration file as an argument
- the configuration file is parsed (an actual configuration object can also be used) and loaded
- two event queue are created as explained below, a high priority queue and a low priority queue
- a `context` is created, based on `kcwidrpframework.models.processing_context.ProcessingContext`
- the event loop is started in a separate thread

The `start()` method of the framework accepts the following parameters: `qm_only` to start the queue system without adding any event, `ingest_data_only` to TBD(ASK SHUI), `wait_for_events` to prevent the framework from exiting after the last event if processed, and `continuous` TBD(ASK SHUI).

3.1.2 The event loop

The core of the framework is an event loop implemented as a thread. This loop performs the simple operations of inspecting the low and high priority queues (described below), picking the first event from the top of the queues, converting the event to an "action", and executing the action.

The sequence of operations is:

- check for events in the high priority queue, if none:
- check for events in the low priority queue
- if no events, either exit or repeat the loop depending on user parameters
- convert the event into an action (an actual class, or a function) using the `event_table` of the pipeline
- if the action is a function, look for preconditions and postconditions associated with it
- if the action is a class, run the `apply()` method of the class, which automatically checks for pre and post conditions

3.1.3 The event queues

The framework uses two event queues that are charged with handling the actual execution of the code. These two queues, called the Low Priority Queue (LPQ) and High Priority Queue (HPQ) are implemented as standard Python Queues.

The reason for having two queues is related to the need to execute sequences of events in the controlled manner. A typical example is the need to run multiple operations (multiple events, resulting in multiple actions) on the same file after it has been ingested. Let's assume that the default event upon ingestion processes the header and, depending on the image type, triggers a chain of events that are specific to the particular image type: the sequence could include a simple set of `remove_overscan` and `trim_overscan` for a bias but very complicated for a science frame (`remove_overscan`, `trim_overscan`, `subtract_bias`, `apply_flat_field`, `apply_lambda_calib` and so on). If a directory contains a number of files, the LPQ would immediately be filled by calls to the ingestion events for each of the file. As the framework starts to process them, the linked events would temporarily fill up the HPQ with the processing steps, and the framework would process them before moving on to the next file. An exception to this is the multiprocessing mode which is described later.

3.1.4 Adding events to the queues

This issue is discussed in other sections such as **events_actions_** and **startup_scripts_** but it is useful to provide a general guide here.

Events can be pushed to the low priority queue by using:

```
framework.append_event('my_event', arguments)
```

Where `arguments` is an instance of the `Arguments` class described in **arguments_**.

While there should be no need to do it, we can also send events directly to the high priority queue, by using:

```
framework.push_event('my_event', argument)
```

Pushing events to the high priority queue can disrupt the correct processing sequence. It is used by the framework itself as part of linking events into recipes as described in **pipelines_**.

3.2 Startup Scripts

The easiest way to start the framework is by using startup scripts. Unfortunately, due to the flexibility of the framework and of the wide variety of different ways of processing files, the start up script is the most complicated piece of the framework.

A complete example of such as script can be found in the `pipeline_template/scripts` directory.

It is convenient to think of this script as being made of different blocks, not all of which are necessary at any given time.

3.2.1 The import block

To use the framework, add these imports to the startup script:

```
from keckdrpframework.core.framework import Framework
from keckdrpframework.config.framework_config import ConfigClass
from keckdrpframework.models.arguments import Arguments
from keckdrpframework.utils.drpf_logger import getLogger
```

The last import is only necessary if we want to customize the logging system, for example to add a secondary log for the pipeline, distinct from the main framework log.

Additional imports that are normally used are:

```
import subprocess
import time
import argparse
import sys
import traceback
import pkg_resources
import logging.config
```

Finally, it is important to import the actual pipeline definition file. The location of the pipeline can be found by looking at the structure of the `pipeline_template` package.

```
from template.pipelines.template_pipeline import TemplatePipeline
```

Of course, this assumes that there is a pipeline package called `template`, that is has a submodule called `pipelines` (basically a directory with a `__init__.py` file) with a `template_pipeline.py` module which describes the actual pipeline as specified in **pipelines_**.

3.2.2 The argument parser

The following section of the startup script implements a function that parses arguments and passes them to the main function.

There is no mandatory structure for the argument parser: the purpose of the startup script is to decide how to start the framework, which files to work on, and which event(s) to trigger. Any way that provides access to these functions is acceptable. A good way to learn about the possible parameters is again to look at the `pipeline_template` startup script, but we can provide a possible set of useful parameters here. It is important to note that if a parameter is offered

to the users, the corresponding piece of code must be added to the main function, which is the reason why we cannot really provide a standardized startup script.

We start with the definition of the function:

```
def _parseArguments(in_args):
    description = "Template pipeline CLI"

    # this is a simple case where we provide a frame and a configuration file
    parser = argparse.ArgumentParser(prog=f"{in_args[0]}", description=description)
```

- `parser.add_argument('-c', dest="config_file", type=str, help="Configuration file")` This parameter can be used

to override the standard configuration file for the pipeline. In the `pipeline_template` package, the configuration files are stored in a `config` directory, and accessed directly using the package discovery system offered by the `pkg_resources` Python module. If we add this parameter, users can build a configuration file in the working directory and then specify that it should be used instead of the default one.

- `parser.add_argument('-f', '--frames', nargs='*', type=str, help='input image file (full path, list ok)', default=None)`

This parameters specifies which files should be processed by the pipeline

- `parser.add_argument('-l', '--list', dest='file_list', help='File containing a list of files to be processed', efault=None)`

If used, this parameters allows users to provide a file containing a list of files to be processed

- `parser.add_argument('-infiles', dest="infiles", help="Input files", nargs="*")` Paired with the next argument (`-d`)

this argument specifies the file pattern to use in ingesting the files in a directory

- `parser.add_argument('-d', '--directory', dest="dirname", type=str, help="Input directory", nargs='?', default=None)`

Used with the previous argument, this argument specifies which directory should be used to ingest files.

- `parser.add_argument('-m', '--monitor', dest="monitor", action='store_true', default=False)` If this flag is

set in the command line, after ingesting all the files in the directory specified, the framework will enter into monitoring mode, and keep ingesting files as they appear. It has to be specified together with the `-W` argument which tells the framework to continue operating even when all the events have been processed.

The following arguments are reserved for the pipeline control flow and will be described separately (TBD!!)

```
parser.add_argument("-i", "--ingest_data_only", dest="ingest_data_only", action=
↪"store_true",
                    help="Ingest data and terminate")
parser.add_argument("-w", "--wait_for_event", dest="wait_for_event", action="store_
↪true", help="Wait for events")
parser.add_argument("-W", "--continue", dest="continuous", action="store_true",
                    help="Continue processing, wait for ever")
parser.add_argument("-s", "--start_queue_manager_only", dest="queue_manager_only",␣
↪action="store_true",
                    help="Starts queue manager only, no processing",
```

Finally, the `_parseArguments` function is closed by passing the results to the main function:

```
args = parser.parse_args(in_args[1:])
return args
```

3.2.3 The main function

Opening

The main function opens with:

```
def main():

args = _parseArguments(sys.argv)
```

Configuration

The configuration system is flexible and can be adapted to the need of the specific pipeline. Normally, it is a good habit to have a separate configuration file for the framework and one for the pipeline itself. We can also have a logging configuration file. In the following we will assume that all three files are used.

The basic principle used is this: a standard configuration file is provided for the framework, the pipeline and the logger. The configuration files live in a `configs` directory, part of the main package defining the pipeline. Since it is the most likely to need changes, the pipeline configuration file can be overridden by the `-c` parameter. A suitable part of the code handles this parameter.

The block looks like this:

```
# START HANDLING OF CONFIGURATION FILES ##########
pkg = 'template'

# load the framework config file from the config directory of this package
# this part uses the pkg_resources package to find the full path location
# of framework.cfg
framework_config_file = "configs/framework.cfg"
framework_config_fullpath = pkg_resources.resource_filename(pkg, framework_config_
↪file)

# load the logger config file from the config directory of this package
# this part uses the pkg_resources package to find the full path location
# of logger.cfg
framework_logcfg_file = 'configs/logger.cfg'
framework_logcfg_fullpath = pkg_resources.resource_filename(pkg, framework_logcfg_
↪file)

# add PIPELINE specific config files
# this part uses the pkg_resource package to find the full path location
# of template.cfg or uses the one defines in the command line with the option -c
if args.config_file is None:
    pipeline_config_file = 'configs/template.cfg'
    pipeline_config_fullpath = pkg_resources.resource_filename(pkg, pipeline_config_
↪file)
    pipeline_config = ConfigClass(pipeline_config_fullpath, default_section='TEMPLATE
↪')
else:
    pipeline_config = ConfigClass(args.pipeline_config_file, default_section='TEMPLATE
↪')
```

While the first two are obvious and are meant to find the full path for the configuration files, the entry for the pipeline deserves some explanation. In the example shown here, we use `ConfigClass`, a class provided by the `keckdrpframework.config.framework_config` module. This class subclasses the standard `ConfigParser` class and provides a set of default parameters and the possibility of specifying a default section of the configuration file. In our case, the section of the configuration file is TEMPLATE. This means that the pipeline configuration file will have a `[TEMPLATE]` section with all the parameters related to the pipeline.

The `pkg` variable should be se to the actual name of the current package.

Initialization of the framework

The framework is initialized by creating an instance of the main class. The class takes two arguments: the first is the pipeline class imported above, the second is either an instance of `ConfigClass` or, in the example shown below, the full path to a configuration file.

```
try:
    framework = Framework(TemplatePipeline, framework_config_fullpath)
    logging.config.fileConfig(framework_logcfg_fullpath)
    framework.config.instrument = pipeline_config
except Exception as e:
    print("Failed to initialize framework, exiting ...", e)
    traceback.print_exc()
    sys.exit(1)
```

In this example, we are making the assumption that this pipeline reduces data for a specific instrument, and to simplify the namespace, we assign the configuration to the variable `instrument`. As a matter of fact, there is only one configuration structure (`framework.config`). This structure can be expanded to contain other configurations, such as the instrument or pipeline configuration shown in the example. This ensures that the main configuration and all the additional configuration are always available for classes and functions. Note also that we are configuring the entire logging system using `logging.config.fileConfig`. The example configuration file for the logger (`configs/logger.cfg`) already defines two different loggers, one for the framework (`DRPF`) and one for the pipeline (`TEMPLATE`). Each of the two loggers use two handlers, a console handler and a file handler. The entire system can be configured as desired by changing the logger configuration file.

The final step in starting the framework is to assign two loggers to their respective objects:

```
framework.context.pipeline_logger = getLogger(framework_logcfg_fullpath, name=
↪"TEMPLATE")
framework.logger = getLogger(framework_logcfg_fullpath, name="DRPF")
```

Processing of files and arguments

This part of the script depends on which parameters are offered to the users in the argument parsers. In the assumption that the parameters are `-f, -l, -i, -d, -m`, the following example shows how to deal with those options.

```
if args.queue_manager_only:
    # The queue manager runs for ever.
    framework.logger.info("Starting queue manager only, no processing")
    framework.start_queue_manager()

# frames processing
elif args.frames:
    for frame in args.frames:
        # ingesting and triggering the default ingestion event specified in the␣
↪configuration file
```

(continues on next page)

(continued from previous page)

```
        framework.ingest_data(None, args.frames, False)

# processing of a list of files contained in a file
elif args.file_list:
    frames = []
    with open(args.file_list) as file_list:
        for frame in file_list:
            if "#" not in frame:
                frames.append(frame.strip('\n'))
    framework.ingest_data(None, frames, False)

# ingest an entire directory, trigger "next_file" on each file, optionally continue␣
↪to monitor if -m is specified
elif (len(args.infiles) > 0) or args.dirname is not None:
    framework.ingest_data(args.dirname, args.infiles, args.monitor)

framework.start(args.queue_manager_only, args.ingest_data_only, args.wait_for_event,␣
↪args.continuous)
```

The code is rather self explanatory. Note that we are making calls to `ingest_data`: this method of the framework is used to perform the basic ingestion and processing of a file. It does two things: 1) parse the header and create a Pandas dataframe with the header keywords as columns and entries for each file and 2) trigger the default ingestion event as specified in the framework configuration file, usually `next_file`.

Users have full control on this startup procedure. For example, it is possible to set the default ingestion event to one of the "do nothing" events, such as `noop` or `no_event` (see the base pipeline class), and then manually trigger a custom event on each of the imported files by doing something like this:

```
for frame in args.frames:
   arguments = Arguments(name=frame)
   framework.append_event('my_preferred_event', arguments)
```

The final step is to protect the main function:

```
if __name__ == "__main__":
main()
```

3.2.4 Operational modes

Reduction of individual files

To reduce a single file or a set of files, the framework can be started with the following command line:

```
>>> template_script.py -frames=file1.fits file2.fits
```

The default script will add a default event to the queue, using the file name as a argument. This event is specified in the configuration file, as `default_ingestion_event`. A standard event is provided as default, called `ingest_only`. This event is always available, inherited from the `BasePipeline`. It does not process the data in any way.

Ingestion of all the files in a specified directory

If a number of files are already stored in a specified directory, the framework can be started with the following command line:

```
>>> template_script.py -infiles=*.fits -directory=/home/data
```

All the files in the specified directory will be ingested if they match the `infiles` pattern, and a `next_file` event will be triggered for each of them. If `-m -W` are specified in the command line, the framework will continue to monitor the directory, and trigger the default ingestion event for each new file. See previous section for a description of the default event.

Starting the framework processing engine with no files

It is possible to start the framework independently from any actual data to process. This is useful for the multiprocessing mode.

To start the framework in this mode, use this command line:

```
>>> template_script.py
```

3.3 Primitives

The operational code is contained in primitives, which are either functions or classes and are limited to simple, single-purpose operations. The framework offers a few basic primitives and a template to build your own. An example of a primitive would be a function that subtract the bias level from a CCD image.

Primitives can be defined in a number of different ways.

3.3.1 Primitives as classes

The best way to define a primitive is by subclassing the `Base_primitive` class, which contains a number of useful methods. In its full implementation, the `Base_primitive` class contains the following methods:

- `_pre_condition`, which must return `True` for the framework to actually execute the code
- `_perform`, which contains the actual code and should return valid arguments
- `_post_condition`, which is checked upon completion of the code contained in the `_perform` method

The execution of the code contained in a class defined following our template is governed by the `apply` method, which is standard and should not be modified. The apply method works like this:

```
def apply (self):
    if self._pre_condition():
        output = self._perform()
        if self._post_condition():
            self.output = output
    return self.output
```

If a the full check on pre and post conditions is not necessary, a simpler class can be defined by simply preserving the `__init__` and the `_perform` methods. An example of a primitive defined this way is a simple FITS reader:

```
def open_nowarning (filename):
   with warnings.catch_warnings():
      warnings.simplefilter('ignore', AstropyWarning)
      eturn pf.open(filename)

class simple_fits_reader (Base_primitive):
```

(continues on next page)

(continued from previous page)

```
    """
    classdocs
    """

    def __init__(self, action, context):
        '''
        Constructor
        '''
        Base_primitive.__init__(self, action, context)

    def _perform (self):
        '''
        Expects action.args.name as fits file name
        Returns HDUs or (later) data model
        '''
        name = self.action.args.name
        self.logger.info (f"Reading {name}")
        out_args = Arguments()
        out_args.name = name
        out_args.hdus = open_nowarning(name)

        return out_args
```

In this case we are using an externally defined function called `open_nowarning` and calling it from within the class. Note that way that the return arguments are build: first, an instance of the `Arguments` class is instantiated, then two properties are defined: `name` and `hdus`. We will describe the use of arguments in the following section.

If the standard class is used to define a primitive, note that the `__init__` method should initialize the `Base_primitive` as well. The example above (the simple FITS reader) shows how to do that, and can be copied and pasted exactly as it is. We will return on the concept of `action` and `context` later in this document.

Note that once the class is defined, it must be made available to the pipeline. This means that the class must be defined in a module or library or regular Python file that can be imported. See the paragraph about directory structure for a suggestion on the proper location for primitives.

3.3.2 Primitives as functions

For simple operations, functions can be used to define primitives instead of classes. In this case, there are no fixed rules as to how the arguments are passed to the function. Functions can be defined directly within the definition of a pipeline (see pagraph about pipelines) or in their own file, as long as the file can be imported and the function added to the namespace.

TBD: describe the pre and post functions

3.4 Arguments

The `Arguments` class provides a flexible way to store any argument that need to be passed to functions or classes. Other pipeline frameworks have adopted a well-defined data model, using arrays or more sophisticated data structures to store FITS files or tables.

For this framework, we have opted for the maximum flexibility.

A very simple argument can be built by using just the file name:

```
from keckdrpframework.models.arguments import Arguments
file_name = 'kb01.fits'
arguments = Arguments(name=file_name)
```

This argument can then be used to add an event to the processing queue:

```
framework.append_event('my_code', arguments)
```

The argument class can be extended with any type of Python object using the syntax `argument.<object`. An example of using this concept is the way that we process the Keck Cosmic Web Imager (KCWI) data. For this instrument, the raw FITS file is made of two extensions: the actual CCD data and a table listing the shutter open/close events with timestamps.

To read this type of FITS files, we can define a specific FITS reader for KCWI as:

```
def kcwi_fits_reader(file):
    """A reader for KeckData objects.
    """
    hdul = fits.open(file)

    if len(hdul) == 2:
        # 1- read the first extension into a ccddata
        ccddata = CCDData(hdul[0].data, meta=hdul[0].header, unit='adu')
        # 2- read the table
        table = hdul[1]

    return ccddata, table
```

We can then create a primitive that can take care of ingesting a KCWI FITS file by subclassing the `Base_primitive` as described in the previous section (see **primitives_**):

```
class kcwi_fits_ingest(Base_primitive):

    def __init__(self, action, context):
        '''
        Constructor
        '''
        Base_primitive.__init__(self, action, context)

    def _perform(self):
        '''
        Expects action.args.name as fits file name
        Returns HDUs or (later) data model
        '''
        name = self.action.args.name
        self.logger.info(f"Reading {name}")
        out_args = Arguments()
        out_args.name = name
        ccddata, table = self.kcwi_fits_reader(name)
        out_args.ccddata = ccddata
        out_args.table = table
        out_args.imtype = out_args.hdus.header['IMTYPE']

     return out_args
```

Note that the name of the file is passed to this function as part of the `action` dictionary. We will describe this dictionary later. The relevant code is the construction of the `out_args` variable: first, it is instantiated as `Arguments`, then it is populated with various elements such as an Astropy NDData CCDData object for the CCD data, and a table.

Finally, and just for convenience, the image type is extracted from the header and assigned to the `imtype` property of the class. Any other property can be added to the arguments.

By doing this, we have effectively defined our own KCWI data model as being composed of a CCDData object, plus a table, plus additional extracted information such as the image type.

3.5 Pipelines

Through the creation of a pipeline, you can make the code that is contained in the primitives available to the framework so that it can be applied to specific arguments.

Pipelines are created by subclassing the `Base_pipeline` class and adding an event table. In its basic implementation, a pipeline would look like this:

```
class MyPipeline(Base_pipeline):
  """
  My own pipeline
  """

  event_table = {
      "my_event_name": ("my_primitive", "my_state", "my_next_event")
      }

  def __init__(self):
      """
      Constructor
      """
      Base_pipeine.__init__(self)
```

This simple construction tells the framework that there is either a class or a function called `my_primitive`, and that it should be run when `my_event_name` is triggered. When that code is running, the state of the framework will be set to `my_state`. After the execution of the code, the framework would go on to trigger `my_next_event` which is currently not defined.

The `event_table` is the core of the pipeline: it makes all the code in your primitives available to the framework through the concept of `events` which we will describe in more detail later.

There is a pre-defined event table that is built into the base pipeline class: it contains the operations that the framework should be doing if no event is present in the execution queue, essentially implementing the infinite loop that the framework executes while waiting for events.

As an example of a more realistic pipeline, here are some examples taken from the KCWI pipeline:

```
event_table = {
       "next_file": ("ingest_file", "file_ingested", "file_ingested"),
       "file_ingested": ("action_planner", None, None),
       # BIAS
       "process_bias": ("process_bias", None, None),
       # CONTBARS PROCESSING
       "process_contbars": ("process_contbars", "contbars_processing_started",
↪"contbar_subtract_overscan"),
       "contbar_subtract_overscan": ("subtract_overscan", "subtract_overscan_started",
↪ "contbar_trim_overscan"),
       "contbar_trim_overscan": ("trim_overscan", "trim_overscan_started", "contbar_
↪correct_gain"),
       "contbar_correct_gain": ("correct_gain", "gain_correction_started", "contbar_
↪find_bars"),
```

(continues on next page)

(continued from previous page)

```
        "contbar_find_bars": ("find_bars", "find_bars_started", "contbar_trace_bars"),
        "contbar_trace_bars": ("trace_bars", "trace_bars_started", None),
        # ARCS PROCESSING
        "process_arc": ("process_arc", "arcs_processing_started", "arcs_subtract_
↪overscan"),
        "arcs_subtract_overscan": ("subtract_overscan", "subtract_overscan_started",
↪"arcs_trim_overscan"),
        "arcs_trim_overscan": ("trim_overscan", "trim_overscan_started", "arcs_correct_
↪gain"),
        "arcs_correct_gain": ("correct_gain", "gain_correction_started", "arcs_extract_
↪arcs"),
        "arcs_extract_arcs": ("extract_arcs", "extract_arcs_started", "arcs_arc_offsets
↪"),
        "arcs_arc_offsets":  ("arc_offsets", "arc_offset_started", "arcs_calc_prelim_
↪disp"),
        "arcs_calc_prelim_disp": ("calc_prelim_disp", "prelim_disp_started", "arcs_
↪read_atlas"),
        "arcs_read_atlas": ("read_atlas", "read_atlas_started", "arcs_fit_center"),
        "arcs_fit_center": ("fit_center", "fit_center_started", None)
        }

def action_planner (self, action, context):
        if action.args.imtype == "BIAS":
            bias_args = Arguments(name="bias_args",
                                  groupid = groupid,
                                  want_type="BIAS",
                                  new_type="MASTER_BIAS",
                                  min_files=context.config.instrument.bias_min_nframes,
                                  new_file_name="master_bias_%s.fits" % groupid)
            context.push_event("process_bias", bias_args)
        elif "CONTBARS" in action.args.imtype:
            context.push_event("process_contbars", action.args)
        elif "FLAT" in action.args.imtype:
            context.push_event("process_flat", action.args)
        elif "ARCLAMP" in action.args.imtype:
            context.push_event("process_arc", action.args)
        elif "OBJECT" in action.args.imtype:
            context.push_event("process_object", action.args)
```

The `next_file` event is triggered when a new file is generated (the description of how to trigger events will be provided later). This triggers the `ingest_file` function, followed by `action_planner`. Note that `action planner` is defined right here, inside the definition of the pipeline itself. This is another option available to users to define functions (see the primitives section): the `action_planner` function has the only purpose of deciding which event should be triggered based on the image type, and as such is logically connected with the initial steps of the pipeline. For this reason, this user found it useful to define it here, rather than in its own library file. From an execution point of view, this makes no difference.

A pipeline definition file should also import the framework package and all the primitives that the user has defined. For example, in the case of KCWI and having created a directory called `primitives`, the import section might look like this:

```
from keckdrpframework.pipelines.base_pipeline import Base_pipeline

from ..primitives.kcwi_primitives import *
```

The relative import of the primitives is based on a specific directory structure which will be discussed later. Any directory structure or packaging system can be used. As long as there is a way to add the primitives to the namespace,

they will be used.

3.6 Events and Actions

3.6.1 Events

A pipeline is defined by creating an event table, linking `events` to the corresponding class or code (see **pipelines_**).

To define an event, we need to specify the following four elements: `name`, `action`, `state`, and `next event`.

The `name` is arbitrary and is the only connection between the pipeline, the framework and the corresponding code. This means that there is no way to directly call a function or a class other than by the `name` associated with it.

The `state` is arbitrary. While not widely used yet, it can be used to control the flow of the pipeline. For now, it is sufficient to know that once an event is triggered, the special variable `context.state` will be set to the value specified in this field.

The `next event` array element is used to create automatic chains of events, if that is desired. For example, if you are creating a basic CCD reduction pipeline, you could write events like this:

```
event_table = {
   "correct_bias":     ("subtract_bias", "bias_processing", "correct_overscan"),
   "correct_overscan: ("fit_and_sub_overscan", "overscan_processing", "correct_flat"),
   "correct_flat":     ("fit_and_div_flat", "flat_processing", None)
   }
```

If `correct_bias` is triggered, the framework would automatically proceed to trigger `correct_overscan`, and continue with `correct_flat`. At this point, the framework would encounter the event `None` and would not proceed further. The variable `context.state` would change value from `bias_processing` to `overscan_processing` to `flat_processing`.

3.6.2 Actions

The `Base_pipeline` offers a method to convert an `Event` into an `Action`.

This operation searches the namespace for classes or functions that match the `action` field of the `event` that has been triggered, and sends the resulting code to the framework for execution.

The actual execution depends on how the code is defined. If the code is contained in a class the framework would look for `pre` and `post` conditions and run the `apply` method if it is defined.

See **primitives_** for further information.

3.7 Plotting with Bokeh

Because the framework runs in a thread, it might be difficult to plot directly with Matplotlib.

A solution is to use Bokeh, which is thread-safe.

An implementation of this solution is offered by the framework using the bokeh_plotting module. In this implementation, we will run a bokeh server, started up by the initialization script, and then connect to it.

To plot with bokeh, start by adding a special event to your pipeline event table:

```
from ..primitives.start_bokeh import start_bokeh

class mypipeline(BasePipeline):
    """
    Generic pipeline

    """

    event_table = {
        "start_bokeh": ("start_bokeh", None, None)
    }
```

The special event is described in a primitive that we will call start_bokeh.py:

```
from keckdrpframework.primitives.base_primitive import BasePrimitive

from bokeh.client import push_session
from bokeh.io import curdoc
from bokeh.plotting.figure import figure
from bokeh.layouts import column



class start_bokeh(BasePrimitive):

    def __init__(self, action, context):
        '''
        Constructor
        '''
        BasePrimitive.__init__(self, action, context)


    def _perform(self):

        self.logger.info("Enabling BOKEH plots")

        self.context.bokeh_session = push_session(curdoc())
        p = figure()
        c = column(children=[p])
        curdoc().add_root(c)
        self.context.bokeh_session.show(c)
```

This special event can be then triggered immediately by your startup script and added to the high priority queue:

```
subprocess.Popen('bokeh serve', shell=True)
framework.append_event('start_bokeh', None)
```

3.7.1 Plotting

The actual plotting is performed by the primitives as needed.

As an example:

3.8 Creating a pipeline using the template

In this example we will create a data reduction package called MyDRP.

The directory `pipeline_template` provides a simple starting point to create a data processing pipeline.

Start by making a copy of the directory with all the included subdirectories

```
>>> mkdir MyPipeline
>>> cd MyPipeline
>>> cp -r <KeckDRPFramework_LOCATION>/pipeline_template/. .
```

3.8.1 Setup.py

You can now start editing the files in the new pipeline, starting with `setup.py`. In this file, it is important to edit the NAME, the description and the licence. Note that this file assumes that any command line interface script will live in the `scripts` directory. Note also that the package name is currently set to `template`. In the next step, we will rename this directory to be the actual package name of your pipeline, so you need to change this variable accordingly.

The name of your pipeline should be set correctly in the `NAME` variable and in the `packages` variable of the `setup` dictionary.

3.8.2 The main pipeline

Defining a pipeline is essentially the same as defining the entries of the `event_table`. A complete description of the event table is provided in *Events and Actions*.

The `import` section of this file is made of two parts: first we import the necessary framework modules such as:

```
from keckdrpframework.pipelines.base_pipeline import BasePipeline
from keckdrpframework.models.processing_context import ProcessingContext
from keckdrpframework.primitives.simple_fits_reader import SimpleFitsReader
```

The `SimpleFitsReader` import is not necessary but it is a good starting point to import FITS files.

The next step is to import primitives that are defined in the `primitives` directory. As explained in the *Primitives* section, if the name of the file containing a primitive corresponds to the name of the class that defines the primitive, there is no need to import it (see the example call to `template2` in the event table). If this is not the case, then the primitive must be imported explicitely (see the example call to `templace` in the event table).

```
from template.primitives.Template import MyTemplate
```

In the simple case in which a single primitive is invoked, a single entry in the event table is all that is needed. Remember that the format for the event table is:

```
event_name: (primitive_name, state, next event)
```

Which can be simplified to:

```
event_name: (primitive_name, None, None)
```

if no state update is required and we don't need to trigger another event after the first.

The template pipeline contains 4 events, which have been chosen to illustrate 4 possible cases of the use of primitives.

- the `next_file` event calls the primitive `SimpleFitsReader` which is a standard primitive provided by the framework and imported explicitly.
- the `template` event calls the primitive `MyTemplate` which is defined in a file called `Template.py`, and imported explicitly. This primitive belongs to this specific pipeline, not to the framework.
- the `template2` event calls the primitive `Template2` which is defined in a file called `Template2.py`. Because the name of the class and the name of file are the same, there is no need to explicitly import the module: the framework will autodiscover it and import it.
- the `template_action` event calls the primitive `template_action`, which is just a function defined in this same pipeline file. This is how we define simple, standard events that don't need their own file or module.

Note that this is a true pipeline, in the sense that each event automatically trigger another one: this is achieved by declaring the next event (3rd element of the tuple) to be the next event in the pipeline: `next_file` calls `template`, which in turns calls `template2`, which calls `template_action`. This is not necessary: this way of building a pipeline simulates the concept of a recipe. It is entirely possible to define a set of independent, disconnected events.

3.8.3 Creating the startup script

The final step to run the pipeline is to trigger eventd and apply it to a file, such as FITS file. There are many ways of doing this (see _startup_script).

Let's analyze the content of the startup script provided as an example.

We start by importing the newly created pipeline:

```
from  template.pipelines.template_pipeline import TemplatePipeline
```

We then define a set of command line arguments in a function that is passed to the argument parser.

```
def _parseArguments(in_args):
  description = "Template pipeline CLI"

  # this is a simple case where we provide a frame and a configuration file
  parser = argparse.ArgumentParser(prog=f"{in_args[0]}", description=description)
  parser.add_argument('-c', dest="config_file", type=str, help="Configuration file")
  parser.add_argument('-frames', nargs='*', type=str, help='input image file (full␣
↪path, list ok)', default=None)

  # in this case, we are loading an entire directory, and ingesting all the files in␣
↪that directory
  parser.add_argument('-infiles', dest="infiles", help="Input files", nargs="*")
  parser.add_argument('-d', '--directory', dest="dirname", type=str, help="Input␣
↪directory", nargs='?', default=None)
  # after ingesting the files, do we want to continue monitoring the directory?
  parser.add_argument('-m', '--monitor', dest="monitor", action='store_true',␣
↪default=False)

  # special arguments, ignore
  parser.add_argument("-i", "--ingest_data_only", dest="ingest_data_only", action=
↪"store_true",
                      help="Ingest data and terminate")
  parser.add_argument("-w", "--wait_for_event", dest="wait_for_event", action="store_
↪true", help="Wait for events")
  parser.add_argument("-W", "--continue", dest="continuous", action="store_true",
                      help="Continue processing, wait for ever")
```

(continues on next page)

(continued from previous page)

```
  parser.add_argument("-s", "--start_queue_manager_only", dest="queue_manager_only",␣
↪action="store_true",
                      help="Starts queue manager only, no processing",
  )

  args = parser.parse_args(in_args[1:])
  return args
```

The next step is to define a `main()` function, which will parse the arguments and start the processing.

The template contains a number of useful comments that should guide the user throughout the process of setting up the specific pipeline.

A concept that deserve some explanation is the triggering of the first event.

The framework configuration file `framework.cfg` contains the definition of the default event that is triggered when a file is ingested, specified as:

```
#
# Default event to trigger on new files
#
default_ingestion_event = "next_file"
```

This means that if we don't make any other choice, and we call the method `framework.ingest_data` on the list of frames, the framework will automatically trigger the `next_file` event on each file specified on the command line or in a specified directory. Because we have this event in our `event_table`, this will work perfectly, and the rest of the events will be triggered in sequence as specified in the `event_table`.

Sometimes, it is desirable to trigger a different event. For example, we can specify a different type of `next_file` which only parses the header but does not trigger any processing. To do so, we would first change the `event_table` to start with:

```
event_table = {

    # this is a standard primitive defined in the framework
    "next_file": ("SimpleFitsReader", "file_ready", None),
```

We would then manually add the desired event to the queue, as part of the `template_script.py`, immediately after the ingestion:

```
elif args.frames:
 for frame in args.frames:
     # ingesting and triggering the default ingestion event specified in the␣
↪configuration file
     framework.ingest_data(None, args.frames, False)
     # manually triggering an event upon ingestion, if desired.
     arguments = Arguments(name=frame)
     framework.append_event('template', arguments)
```

In this case, for each file we automatically trigger `next_file`, which returns the control to the framework without triggering anything else. After that, we define a new argument based on the name of the file, and we manually add the `template` event to the queue. The result is exactly the same as before, but we have much more control on what happens.

If instead of providing a list of files we want to process an entire directory, we can use the `-d` option paired with the `-i` option, to specify the directory and the file pattern to use. If we want to continue monitoring the directory for new files, we can use the `-m -W` combination.

3.8.4 Installation and examples

To install the pipeline, use:

```
python setup.py develop (or install)
```

A few example of using the template pipeline on a set of test data is provided here:

```
> template_script -f <KeckDRPFramework_LOCATION>/keckdrpframework/unit_tests/test_
↪files/*.fits

> template_script -d <KeckDRPFramework_LOCATION>/keckdrpframework/unit_tests/test_
↪files -i *.fits -m -W
```

CHAPTER 4

Reference API

4.1 Reference/API

Basic framework module

This is the main module implementing the framework.

@author: skwok

class keckdrpframework.core.framework.**Framework**(*pipeline_name*, *configFile*)

This class implements the core of the framework.

The processing is event driven. An event can be defined as a change in set of items of interest, for example files or directories, or something in memory. Events are appended to a queue. Events are associated with arguments, such time, name of files, or new values of some variables. In a loop, an event is taken out from the queue and translated into an action.

An action is a call to a regular function, a method in the pipeline class or in a regular class.

If desired, a Data_set() can created to keep a list of files in memory.

config

Instance of ConfigClass that uses the configuration file to create a set of configuration parameters

Type ConfigClass

logger

Type log

pipeline

pipeline can be a string, a module, a class or an object of subclass base_pipeline The pipeline that will be used in the framework

Type pipeline

context

Instance of Processing_context, which passed along to all processing steps.

pipeline_name: name of the pipeline class containing recipes

Creates the event_queue and the action queue

append_event(*event_name*, *args*, *recurrent=False*)
Appends low priority event to the end of the queue

end()
Releases the event_queue. Needed when a client ingest_data and then quits.

event_to_action(*event*, *context*)
Returns an Action() Passes event.args to action.args. Note that event.args comes from previous action.output.

This method is called in the action loop. The actual event_to_action method is defined in the pipeline and it depends on the incoming event and context.state.

execute(*action*, *context*)
Executes one action The input for the action is in action.args. The action returns action_output and it is passed to the next event if action is successful.

get_event()
Retrieves and returns an event from the queues. First it checks the high priority queue, if fails then checks the regular event queue.

If there are no more events, then it returns the no_event_event, which is defined in the configuration.

ingest_data(*path=None*, *files=None*, *monitor=False*)
Adds files to the data_set. The data_set resides in the framework context.

init_signal()
Captures keyboard interrupt

main_loop()
This is the main action loop.

This method can be called directly to run in the main thread. To run in a thread, use start_action_loop().

on_exit(*status=0*)
Hook fo exit Subclasses can override to continue in the main_loop or call exit(status)

on_state(*state*)
Hook to change context state. Default is to ignore state = 'stop'. To terminate, override this method to return 'stop'.

start_action_loop()
This is a thread running the action loop.

wait_for_ever()
Because the action loops runs in a thread, this methods waits until keep_going is false.

keckdrpframework.core.framework.**create_context**(*event_queue=None*, *event_queue_hi=None*, *logger=None*, *config=None*)
Convenient function to create a context for working withtout the framework. Useful in Jupyter notebooks for example.

Python Module Index

k

Index