Recipe-primitives Tutorial

Release 0.1

Gemini

April 22, 2010

Contents

1 Recipe system and primitives

What is a recipe?

A recipe is a text file containing a list of sequential instructions called 'Primitives', for example a file 'recipe.myGmos' can have:

```
# Recipe for the prototype presentation
getProcessedBias
prepare
biasCorrect
mosaicChips
display(displayID = single)
shift
measureIQ
setStackable
display(displayID = combined)
measureIQ
```

You can have comments (#) only in separated lines. Mostly all primitives would not have arguments in the recipes.

What is a primitive?

A primitive is a Python function name that provides a data processing step called by a particular Recipe. A primitive is defined within a Python file as part of a Python Class. These Classes are defined for a particular instrument and dataset type; e.g. 'primitives_GMOS_IMAGE.py'. Some primitives can have generic functionality like “ShowInputs” and as such they should be defined at a higher level in the primitives hierarchy; for example in ‘primitives_GEMINI.py’ which applies to most dataset types.

Dataset types

‘dataset types’ are Astrodata classification names for a particular GEMINI FITS file, regarding which instrument it comes from and processing state is in. For example: [’GEMINI_NORTH’, ‘GEMINI’, ‘IMAGE’, ‘GMOS_N’, ‘GMOS_IMAGE’, ‘GMOS’, ‘GMOS_RAW’, ‘RAW’, ‘UNPREPARED’]

Parameters file for a primitive

A default set of arguments for a given primitive can be store in a parameters file. For example: ‘RECIPES_gm/primitives_GMOS_IMAGE.py’ is the name and location of a primitive for GMOS_IMAGE types. Its parameters file need to be: ‘RECIPES_gm/parameters_GMOS_IMAGE.py’. See below for syntax details.

Primitive example: ‘primitives_GMOS_IMAGE.py’ contains:

```
def ADU2electron(self, rc):        # The only argument in a primitive
                                   # is 'rc' (Reduction Context)
    try:
        files = []
        for ff in rc.inputs:
            files.append(ff.filename)

        ADUToElectron(files, rc['odir'], rc['oprefix'])
    except:
        raise GMOS_IMAGEException("Problem with ADU2electron")
    yield rc
```

Parameter file example: ‘parameters_GMOS_IMAGE.py’ contains:

```
# This is the parameter file for the primitive 'ADU2electron'
# 2 arguments can be passed to the primitive via the 'rc'
# (reduction context) and you can change their values and properties
# using this parameter files. The arguments are 'odir' and 'oprefix'.

# Parameters definition is a Python Dictionary. Its 'keys' are the
# primitives names in the primitives Python file.

localParameterIndex = {   # Start defining default values and properties.
      'ADU2electron':     # This is the primitive name; is a dictionary
                          # with keys as its argument names and their
                          # values -a dictionary also, have the
                          # 'default' value plus properties.
         {'odir'   :{'default':'/tmp',
                     'recipeOverride':True,
                     'userOverride':True,
                     'uiLevel': 'UIADVANCED',
                     'type' : str
                    },
          'oprefix':{'default': 'elex_'
                     'recipeOverride': True,
                     'userOverride': False
                    },
         }
```

1.1 Running a Recipe

Recipes are driven by the content of the input FITS files(s) or in Astrodata context by its ‘dataset type’. At Gemini, we write a recipe for each instrument, dataset type and processing state.

To run a recipe we type ‘reduce’ as a unix command follow by list of FITS files, as in:

```
# This command will not be necessary once the Astrodata system
# is available on the server.
alias reduce <>/astrodata/scripts/reduce.py   # Where the script is located

reduce N20020214S061.fits N20020214S062.fits ... N20020214S069.fits

OR

reduce @input_list.txt              # Read the FITS files from an @ list.
reduce /data/myFits/@test.list      # If file list is in another directory.
```

Before ‘reduce’ read the input files, it will scan the RECIPE_Gemini directory, make a list of the recipes, primitives and parameters. Once a FITS file is read, its dataset type will determine which recipe to run and the primitives in it belongings to the dataset.

For further details, please see the section “Write and use your own recipe/primitive”

1.2 Parameters file

Each primitive can have a default and a local parameters file. Both have different goals so they are written in different ways. The local parameter file is a text file with arbitrary name and can be fed to the primitives as a flag in the ‘reduce’ command; see example below.

Default Parameters File

The naming is the same as primitives except that we use ‘**parameters_**‘ instead of ‘**primitives_**‘; e.g. ‘parameters_GMOS_IMAGE.py’. Notice that this is a regular Python file.:

```
# File: parameters_GMOS_IMAGE.py

# Parameters definition is a Python Dictionary.

localParameterIndex = {    # Start defining default values and properties
      'ADU2electron':      # This is the primitive name
         {'odir'   :{'default':'/tmp',
                     'recipeOverride':True,
                     'userOverride':True,
                     'uiLevel': 'UIADVANCED',
                     'type' : str
                    },
          'oprefix':{'default': 'elex_'
                     'recipeOverride': True,
                     'userOverride': False
                    },
         }
```

Local parameters file.

If we want to overwrite the default values above we can define in a text file the new values. This takes effect only when included as a flag value in the reduce’ command:

```
# File: 'myLocal.par' has:

[GMOS_IMAGE]           # Dataset type
[ADU2electron]         # primitive name
oprefix=fff_           # Reset 'oprefix' value
odir=/tmp/ddd
```

To use this we type:

```
reduce -f myLocal.par N20020214S061.fits
```

Recipe and Primitives Naming convention and location

The Recipe system has a directory named "RECIPES_Gemini" with a set of recipes for different instruments and modes that are executed according to the current input dataset types.

The recipe names need to have a prefix "recipe.".

The primitives and their corresponding parameter files need to have the prefix '**primitives_**' and '**parameters_**'.

A subdirectory 'primitives' in found here with names amongst others: primitives_GEMINI.py, primitives_GMOS_IMAGE.py, parameters_GEMINI.py and parameters_GMOS_IMAGE.py. The content of these are a set of function that applies to corresponding dataset types.

1.3 Write and use your own recipe

For the Recipe system to handle your own recipes and primitives you need the have a directory name with prefix '**RECIPES_**'; for example:

1. mkdir RECIPES_Mytest
2. cd RECIPES_Mytest
3. Create a file 'recipe.mygmos' and type primitives names -assuming they will be used for GMOS_IMAGE Astrodata type:

   ```
   showInputs
   ADU2electron
   ```

   'Reducing' with this recipe:

   reduce -r recipe.mygmos N20020214S061.fits

   If the dataset type of this FITS file is 'GMOS_IMAGE', then the primitive 'showInputs' from the file 'primitives_GEMINI.py' will run. The primitive 'ADU2electron' from 'primitives_GMOS_IMAGE.py' will then run.

4. If you are not using '-r'.

   To run the example in 3) without using the '-r' flag, you need to create or edit the local file: 'recipeIndex.TEST.py' to direct the recipe system to read your own file 'recipe.mygmos'.:

   ```
   # adds recipe 'recipe.mygmos' to the global list
   # for datasets type GMOS_IMAGE.

   localAstroTypeRecipeIndex = {
                               'GMOS_IMAGE': ['mygmos'],
                              }
   ```

```
# Notice that 'mygmos' is the extension to the recipe
# name 'recipe.mygmos'.
```

'Reducing':

```
reduce N20020214S061.fits

How 'reduce' works?

-- Looks for 'RECIPES_...' directories and under these,
   for 'primitives_...' and 'parameters_...' files.

-- Look for a 'recipeIndex.<...>.py' Python file containing

   # adds recipes to the global list for a type

   localAstroTypeRecipeIndex = {'GMOS_IMAGE': ['mygmos'],
                                'NICI_IMAGE': ['niciIm']
                                }

   If a GMOS FITS file is read and is of type GMOS_IMAGE,
   the 'recipe.mygmos' gets executed.
```

1.4 Write and use your own primitive

In your same local directory 'RECIPES_Mytest', create or edit the file 'primitives_GMOS_IMAGE.py'. For sake of example the input FITS files in 'reduce' will have dataset types 'GMOS_IMAGE'.

A Primitive is a member function of the Python Class GMOS_IMAGEPrimitives(GEMINIPrimitives)

Edit your 'primitives_GMOS_IMAGE.py':

```
#####  NOTE
  Need to include the Preamble code defining the GMOS_IMAGEException
  and the GMOS_IMAGEPrimitives classes.

  Please see link right after this block>
#####

def ADU2electron(self, rc):
    """
      This primitive uses the local function 'ADUToElectron'.
      Arguments:
      rc['odir']: Output directory name
      rc['oprefix']: Prefix to the output filenames
    """

    try:
        files = []             # Empty list to put input files
        for ff in rc.inputs:   # rc.inputs contains the filenames
            files.append(ff.filename)

        ADUToElectron(files, rc['odir'], rc['oprefix'])

        # Form the ouput files generated by ADUToElectron. These
        # can be use by the next primitive in the recipe.mygmos
        ofiles=[]
        for ff in rc.inputs:
            stt = oprefix + os.path.basename(ff.filename)
            file = os.path.join(odir, stt)
            ofiles.append(file)
        rc.reportOutput(ofiles)  # Keep these filenames for use in the
                                 # primitive of the recipe if necessary.

    except:
        raise GMOS_IMAGEException("Problem with ADU2electron")

    yield rc

# The 'ADUToElectron' function is given as part of this
# tutorial elsewhere.
```

The preamble code is *here*.

The source code for 'ADUToElectron' is *here*.

Reducing:

reduce -r RECIPES_Mytes/recipe.mygmos N20020214S061.fits

The first primitive in 'recipe.mygmos' 'showInputs' will be processed regardless of the dataset type. If the input FITS file has a dataset type GMOS_IMAGE then the primitive 'ADU2electron' defined above will be process instead of the default one defined elsewhere.

Parameters

Write your 'parameters_GMOS_IMAGE.py' as the example above to define the default values for the local 'ADUtoElectron' function arguments 'odir' and 'oprefix'.

# Use -p to change a parameter value in the primitive:

```
reduce -r RECIPES_Mytes/recipe.gmosmf \
       -p GMOS_IMAGE:ADU2electron:oprefix='recL_' \
          N20020214S061.fits N20020214S061xx.fits

# notice the notation
# ASTRODATATYPE:primitiveName:parameterName=value
```

1.5 More on Primitives

The recipe system allows you to stack a set of input FITS files into distinct stacks based on the values of the header keywords 'OBSID' and 'OBJECT'.

Here is a generic example on how to use Stacks in primitives:

```
# The input filenames comes in rc.inputs

from pyraf.iraf import fitsutil      # Import iraf package

tempset = set()          # We will put the different lists in a Python set.

# Generate the set based on the input list
for inp in rc.inputs:
     tempset.add( IDFactory.generateStackableID(inp.ad))
```

```
    for stackID in tempset:

        # Get a list for each of the stacks
        stacklist = rc.getStack(stackID).filelist

        print stackID, stacklist

        if len( stacklist ) > 1:
            # We can create an '@' file in /tmp
            atfile = rc.makeInlistFile('/tmp/files.lis', stacklist)
            fitsutil.fxheader( atfile)

            # Or just pass the list of files, separated by commas

            fitsutil.fxheader(','.join(stacklist))
```

# Preamble code for primitives_GMOS_IMAGE from primitives_GEMINI import GEMINIPrimitives

from adu2e import ADUToElectron

import os

class GMOS_IMAGEException:

    “”“ This is the general exception the classes and functions in the Structures.py module raise.

    “”“

    def __init__(self, msg=”Exception Raised in Recipe System”):

        “”“This constructor takes a message to print to the user.”“”

        self.message = msg

    def __str__(self):

        “”“ This str conversion member returns the message given by the user (or the default message) when the exception is not caught.

        “”“

        return self.message

class GMOS_IMAGEPrimitives(GEMINIPrimitives):

    def init(self, rc):

        GEMINIPrimitives.init(self, rc)

        return rc

    def ADU2electron(self, rc):

    # put your code here

# ADU2Electron code

```
#! /usr/bin/env python

from astrodata.AstroData import AstroData, prepOutput
import os
from copy import deepcopy

def ADUToElectron(filelist, odir, oprefix):
```

```
"""
 This is a function to convert the ADU counts to electrons
 by multiply the pixel values by the gain.
 Arguments:
   filelist: A python list of FITS filenames
   odir:     Directory pathname for output FITS files
   oprefix:  Prefix for output filenames. Example: If input filename
             is 'S20100323S0012.fits' and 'oprefix' is 'n', the output
             name will be 'nS20100323S0012.fits'
"""

# Loop through the files in filelist
for filename in filelist:
    # Open the file as an AstroData object
    adinput = AstroData(filename, mode='readonly')

    # Verify whether the data has already been converted to electrons
    if adinput.phuValue('ELECTRON') != None:
        print "WARNING: File %s has already been converted to electrons"\
               % filename
        return

    outputname = oprefix + os.path.basename(filename)
    ofile = os.path.join(odir,outputname)
    if os.access(ofile, os.F_OK): os.remove(ofile)

    # Prepare a new output
    #    Propagate PHU and MDF (if applicable) to output.
    #    No pixel extensions yet.
    #    Set output file name.
    #    No overwrite allowed. (default mode for prepOutput)
    #
    # prepOutput copies the adinput PHU and set the name of the new
    # file represented by adout to ofile.

    adout = prepOutput(adinput, ofile)

    # Get the gain values to apply
    # adinput.gain() returns a list, one value for each science extension.
    gain = adinput.gain()

    # Use the deepcopy function to create a true copy and ensure that
    # the original is not modified.s

    adc = deepcopy(adinput)

    # Multiply each science extension by the gain.
    # Append new extension to already prepared output.
    for extension,g,xn in zip(adc, gain, range(len(gain))):
        extension.data = extension.data * g

        adout.append(data=extension.data, header=extension.header)

    # Update PHU with timestamps
    adout.phuSetKeyValue('ELECTRON', fits_utc(),
        comment='UT Modified with convertToElectrons')
    adout.phuSetKeyValue('GEM-TLM', fits_utc(),
        comment='UT Last modification with GEMINI')
        # Write to disk.  The filename was specified when
        # prepOutput was called.
        adout.write()

        # Close files
        adout.close()
        adc.close()
        adinput.close()

import time
def fits_utc():
    """
      Return a UTC string in FITS format:
      YYYY-MM-DDThh:mm:ss
    """

   gmt = time.gmtime()
   time.asctime(gmt)
   fitsT = '%d-%02d-%02dT%02d:%02d:%02d' % gmt[:6]

   return fitsT

if __name__ == "__main__":

    import optparse

    VERSION = '1.0'

    # Parse input arguments
    usage = 'usage: %prog [options] file1 .. fileN'
    p = optparse.OptionParser(usage=usage, version='v'+VERSION)
    p.add_option('--oprefix', '-p', action='store', dest='prefix', default='elec_',
        help='Prefix for the output files')
    p.add_option('--odir', action='store', default='', help='Output directory pathname')

    (options, args) = p.parse_args()

    ADUToElectron(args, options.odir, options.prefix)
```