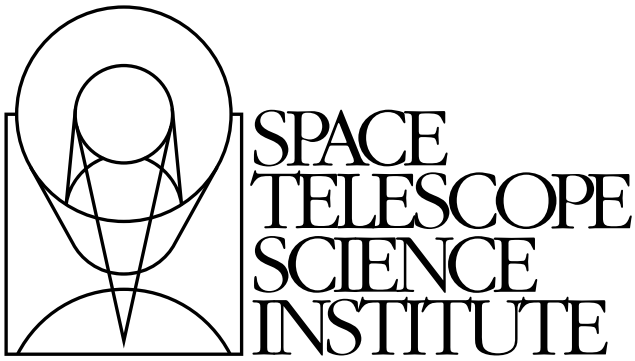


crds Documentation

Release 0.1

STScI

August 15, 2012

CONTENTS

1 Welcome **1**

2 Installation **3**
- 2.1 Getting the Source Code . . . 3
- 2.2 Setting up your Environment . . . 3
- 2.3 Run the Install Script . . . 4
- 2.4 Test the installation . . . 4
- 2.5 Package Overview . . . 4
- 2.6 Dependencies . . . 5

3 Top Level Use **7**
- 3.1 crds.getreferences() . . . 7
- 3.2 crds.get_default_context() . . . 8

4 Non-Networked Use **9**
- 4.1 Overview of Features . . . 9
- 4.2 Important Modules . . . 9
- 4.3 Basic Operations on Mappings . . . 10
- 4.4 Certifying Files . . . 12
- 4.5 Mapping Checksums . . . 13
- 4.6 Which Mappings Use this File? . . . 13
- 4.7 Finding Matches for a Reference in a Context . . . 14

5 About Mappings **15**
- 5.1 Naming . . . 15
- 5.2 Basic Structure . . . 16
- 5.3 Pipeline Mappings (.pmap) . . . 16
- 5.4 Instrument Mappings (.imap) . . . 16
- 5.5 Reference Mappings (.rmap) . . . 17
- 5.6 Selectors . . . 18

CHAPTER

ONE

WELCOME

release-notes

Welcome to the Calibration and Reference Data System (CRDS). CRDS manages the reference files and the rules which are used to assign appropriate references to particular data sets. CRDS has a variety of aspects:

- CRDS is a Python package usable within the context of the STSCI Python environment. The core library can be used without network access.
- CRDS has an interactive web server which assists with the creation and submission of "official" reference files and CRDS mappings (rules). CRDS has a database which tracks delivery metadata for references and mappings.
- The CRDS web server also provides network services via JSONRPC related to best reference determination and file distribution.
- CRDS has a Python client package which accesses the network services to determine and locally cache the best references for a dataset.

CHAPTER

TWO

INSTALLATION

2.1 Getting the Source Code

At this stage of development, installing CRDS is accomplished by checking CRDS source code out from subversion:

```
% svn co https://subversion.assembla.com/svn/crds/trunk  crds
% cd crds
```

2.2 Setting up your Environment

The CRDS checkout has a template file for the C-shell which defines environment variables, env.csh. For JWST development, there are reasonable defaults for everything so you may not need to set these at all. For HST, at a minimum CRDS_SERVER_URL must be defined.

2.2.1 Basic Environment

- **CRDS_PATH** defines a common directory tree where CRDS reference files and mappings are stored. CRDS_PATH defaults to "./crds". Mappings are stored in ${CRDS_PATH}/mappings/[hst|jwst]. Reference files are stored in ${CRDS_PATH}/references/[hst|jwst].
- **CRDS_SERVER_URL** defines the base URL for accessing CRDS network services. CRDS_SERVER_URL defaults to the jwst test server.

2.2.2 Advanced Environment

- **CRDS_MAPPATH** can be used to override CRDS_PATH and define where only mapping files are stored. If mappings are pre-installed, the directory pointed to by CRDS_MAPPATH can be readonly. CRDS_MAPPATH defaults to ${CRDS_PATH}/mappings.
- **CRDS_REFPATH** can be used to override CRDS_PATH and define where only reference files are stored. If references are pre-installed, the directory pointed to by CRDS_REFPATH can be readonly. CRDS_REFPATH defaults to ${CRDS_PATH}/references.
- **CRDS_CFGPATH** can be used to override CRDS_PATH and define where only server configuration information is cached. The directory pointed to by CRDS_CFGPATH should be writable. If CRDS is running in server-less mode, this path is irrelevant. CRDS_REFPATH defaults to ${CRDS_PATH}/config.

- **CRDS_MODE** defines whether CRDS should compute best references using client software (local), server software (remote), or intelligently "fall up" to the server only when the client is deemed obsolete or the server cannot be reached (auto). The default is auto.
- **CRDS_CONTEXT** is an override naming the CRDS pipeline mapping (.pmap) used for computing best references. Ordinarily, CRDS will contact the server to determine the operational pipeline mapping. If the server cannot be reached, CRDS will look in CRDS_CFGPATH to determine the last pipeline context the server recommended. If there is no prior server info available in the cache, CRDS will fall-back to using the default pre-installed mappings, e.g. jwst.pmap. When CRDS_CONTEXT is set, CRDS will ignore server recommendations and availability and use the specified pipeline mapping. However, CRDS_CONTEXT will only be used when `context` was specified to getreferences() as None. If `context` was explicitly specified in a call to getreferences() and was not None, the specified context will override CRDS_CONTEXT. This enables the implementation of command line switches which supercede CRDS_CONTEXT.

Edit env.csh according to your preferences for where to put CRDS files. Then source env.csh to define the variables in your environment:

```
% source env.csh
```

2.3 Run the Install Script

CRDS is partitioned into 3-4 Python packages each of which has it's own setup.py script. To make things easier, the top level directory has a single "install" script which runs all the individual setup.py scripts for you:

```
 % ./install
Installing lib
Installing client
Installing hst
Installing jwst
Installing tobs
final status 000000
```

2.4 Test the installation

CRDS client testing operates locally and does not require access to the server. Basic CRDS client testing can be done as follows:

```
 % ./runtests
 ... copious test output...
----------------------------------------------------------------------
Ran 59 tests in 13.749s

OK
```

2.5 Package Overview

From the perspective of an end user, CRDS consists of 3 or more Python packages which implement different capabilities:

- **crds**
  - core package enabling local use and development of mappings and reference files.

- **crds.client**
  - network client library for interacting with the central CRDS server.
- **crds.hst**
  - observatory personality package for HST, with initial mappings for bootstrapping CRDS and defining how HST files are named, located, and certified.
- **crds.jwst**
  - analogous to crds.hst, for JWST.

2.6 Dependencies

CRDS was developed in and for an STSCI Python environment suitable for pipeline processing. CRDS requires these additional packages to be installed in your Python environment:

- numpy
- pyfits

CHAPTER

THREE

TOP LEVEL USE

This section describes the formal top level interfaces for CRDS intended as the main entry points for the calibration software or basic use. Functions at this level should be assumed to require network connectivity with the CRDS server.

To function correctly, these API calls may require the user to set the environment variables CRDS_SERVER_URL and CRDS_PATH. See the section on *Installation* for more details.

3.1 crds.getreferences()

Given dataset header containing parameters required to determine best references, and optionally a specific .pmap to use as the best references context, and optionally a list of the reference types for which reference files are to be determined, getreferences() will determine best references, cache them on the local file system, and return a mapping from reference types to reference file paths:

```
def getreferences(parameters, reftypes=None, context=None, ignore_cache=False):
    """Return the mapping from the requested `reftypes` to their
    corresponding best reference file paths appropriate for a dataset
    described by `parameters` with CRDS rules defined by `context`::

    parameters :    A mapping of parameter names to parameter value
            strings for parameters which define best reference file matches.

            { str  :   str, int, float, bool }

           e.g.  {
                   'INSTRUME' : 'ACS',
                   'CCDAMP' : 'ABCD',
                   'CCDGAIN' : '2.0',
                   ...
                 }

    reftypes :    A list of reference type names.   For HST these are the keywords
                  used to record reference files in dataset headers.   For JWST,  these
                  are the identifiers which will appear in instrument contexts and
                  reference mappings.

                 e.g.  [ 'darkfile', 'biasfile']

                 If reftypes is None,  return all reference types defined by
                 the instrument mapping for the instrument specified in
                 `parameters`.
```

```
    context :   The name of the pipeline context mapping which should be
            used to define best reference lookup rules,  or None.  If
            'context' is None,  use the latest operational pipeline mapping.

            str

            e.g. 'hst_0037.pmap'

    ignore_cache :   If True,  download all required mappings and references
            from the CRDS server.  If False,  download only those files not
            already in the local caches.

    Returns
    -------
            a mapping from reftypes to cached best reference file paths.

            { str : str }

            e.g.   {
                'biasfile' : '/path/to/file/hst_acs_biasfile_0042.fits',
                'darkfile' : '/path/to/file/hst_acs_darkfile_0056.fits',
                }
    """
```

3.2 crds.get_default_context()

get_default_context() returns the name of the pipeline mapping which is currently in operational use. When no

The default context defines the matching rules used to determine best reference files for a given set of parameters:

```
def get_default_context():
    """Return the name of the latest pipeline mapping in use for processing
    files.

    Returns
    -------
    pipeline context name

        e.g.   'hst_0007.pmap'
    """
```

CHAPTER

FOUR

NON-NETWORKED USE

This section describes using the core crds package without access to the network. Using the crds package in isolation it is possible to develop and use new reference files and mappings. Note that a default install of CRDS will also include crds.client and crds.hst or crds.jwst. In particular, the observatory packages define how mappings are named, where they are placed, and how reference files are checked.

4.1 Overview of Features

Using the crds package it's possible to:

- Load and operate on rmaps
- Determine best reference files for a dataset
- Check mapping syntax and verify checksum
- Certify that a mapping and all it's dependencies exist and are valid
- Certify that a reference file meets important constraints
- Add checksums to mappings
- Determine the closure of mappings which reference a particular file.

4.2 Important Modules

There are really two important modules which anyone doing low-level and non- networked CRDS development will first be concerned with:

- **crds.rmap module**
  - **defines classes which load and operate on mapping files**
    - Mapping
    - PipelineContext (.pmap)
    - InstrumentContext (.imap),
    - ReferenceMapping (.rmap)
  - **defines get_cached_mapping() function**

  - loads and caches a Mapping or subclass instances from files, typically this is a recursive process loading pipeline or instrument contexts as well as all associated reference mappings.
  - this *cache* is an object cache to speed up access to mappings, not the file *cache* used by crds.client to avoid repeated network file transfers.

- **crds.selectors module**

  – defines classes implementing best reference logic

  - MatchSelector
  - UseAfterSelector
  - Other experimental Selector classes

4.3 Basic Operations on Mappings

4.3.1 Loading Rmaps

Perhaps the most fundamental thing you can do with a CRDS mapping is create an active object version by loading the file:

```
% python
>>> import crds.rmap as rmap
>>> hst = rmap.load_mapping("hst.pmap")
```

The load_mapping() function will take any mapping and instantiate it and all of it's child mappings into various nested Mapping subclasses: PipelineContext, InstrumentContext, or ReferenceMapping.

Loading an rmap implicitly screens it for invalid syntax and requires that the rmap's checksum (sha1sum) be valid by default.

Since HST has on the order of 70 mappings, this is a fairly slow process requiring a couple seconds to execute. In order to speed up repeated access to the same Mapping, there's a mapping cache maintained by the rmap module and accessed like this:

```
>>> hst = rmap.get_cached_mapping("hst.pmap")
```

The behavior of the cached mapping is identical to the "loaded" mapping and subsequent calls are nearly instant.

4.3.2 Seeing Referenced Names

CRDS Mapping classes all know how to show you the files referenced by themselves and their descendents. The ACS instrument context has a reference mapping for each of it's associated file kinds:

```
>>> acs = rmap.get_cached_mapping("hst_acs.imap")
>>> acs.mapping_names()
['hst_acs.imap',
 'hst_acs_idctab.rmap',
 'hst_acs_darkfile.rmap',
 'hst_acs_atodtab.rmap',
 'hst_acs_cfltfile.rmap',
 'hst_acs_spottab.rmap',
 'hst_acs_mlintab.rmap',
```

```
 'hst_acs_dgeofile.rmap',
 'hst_acs_bpixtab.rmap',
 'hst_acs_oscntab.rmap',
 'hst_acs_ccdtab.rmap',
 'hst_acs_crrejtab.rmap',
 'hst_acs_pfltfile.rmap',
 'hst_acs_biasfile.rmap',
 'hst_acs_mdriztab.rmap']
```

The ACS atod reference mapping (rmap) refers to 4 different reference files:

```
>>> acs_atod = rmap.get_cached_mapping("hst_acs_atodtab.rmap")
>>> acs_atod.reference_names()
['j4d1435hj_a2d.fits',
 'kcb1734hj_a2d.fits',
 'kcb1734ij_a2d.fits',
 't3n1116mj_a2d.fits']
```

4.3.3 Computing Best References

The primary function of CRDS is the computation of best reference files based upon a dictionary of dataset metadata. Hence, both an InstrumentContext and a ReferenceMapping can meaningfully return the best references for a dataset based upon a parameter dictionary. It's possible define a header as any Python dictionary provided you have sufficient knowledge of the parameters:

```
>>>  hdr = { ... what matters most ... }
```

On the other hand, if your dataset is a FITS file and you want to do something quick and dirty, you can ask CRDS what dataset metadata may matter for determining best references:

```
>>> hdr = acs.get_minimum_header("test_data/j8bt05njq_raw.fits")
{'CCDAMP': 'C',
 'CCDGAIN': '2.0',
 'DATE-OBS': '2002-04-13',
 'DETECTOR': 'HRC',
 'FILTER1': 'F555W',
 'FILTER2': 'CLEAR2S',
 'FW1OFFST': '0.0',
 'FW2OFFST': '0.0',
 'FWSOFFST': '0.0',
 'LTV1': '19.0',
 'LTV2': '0.0',
 'NAXIS1': '1062.0',
 'NAXIS2': '1044.0',
 'OBSTYPE': 'IMAGING',
 'TIME-OBS': '18:16:35'}
```

Here I say *may matter* because CRDS is currently dumb about specific instrument configurations and is returning metadata about filekinds which may be inappropriate.

Once you have your dataset parameters, you can ask an InstrumentContext for the best references for *all* filekinds for that instrument:

```
>>> acs.get_best_references(hdr)
```

{'atodtab': 'kcb1734ij_a2d.fits',
'biasfile': 'm4r1753rj_bia.fits', 'bpixtab': 'm8r09169j_bpx.fits', 'ccdtab': 'o1515069j_ccd.fits', 'cfltfile': 'NOT FOUND n/a', 'crrejtab': 'n4e12510j_crr.fits', 'darkfile': 'n3o1059hj_drk.fits', 'dge-

ofile': 'o8u2214mj_dxy.fits', 'flshfile': 'NOT FOUND n/a', 'idctab': 'p7d1548qj_idc.fits', 'imphttab': 'vbb18105j_imp.fits', 'mdriztab': 'ub215378j_mdz.fits', 'mlintab': 'NOT FOUND n/a', 'oscntab': 'm2j1057pj_osc.fits', 'pfltfile': 'o3u1448rj_pfl.fits', 'shadfile': 'kcb1734pj_shd.fits', 'spottab': 'NOT FOUND n/a'}

In the above results, FITS files are the recommended best references, while a value of "NOT FOUND n/a" indicates that no result was expected for the current instrument mode as defined in the header. Other values of "NOT FOUND xxx" include an error message xxx which hints at why no result was found, such as an invalid dataset parameter value or simply a matching failure.

You can ask a ReferenceMapping for the best reference for single the filekind it manages:

```
>>> acs_atod.get_best_ref(hdr)
>>> 'kcb1734ij_a2d.fits'
```

Often it is convenient to simply refer to a pipeline/observatory context file, and hence PipelineContext can also return the best references for a dataset, but this is really just shorthand for returning the best references for the instrument of that dataset:

```
>>> hdr = hst.get_minimum_header("test_data/j8bt05njq_raw.fits")
>>> hst.get_best_references(hdr)
... for this hdr, same as acs.get_best_references(hdr) ...
```

Here it is critical to call get_minimum_header on the pipeline context, hst, because this will make it include the "INSTRUME" parameter needed to choose the ACS instrument.

4.4 Certifying Files

CRDS has a module which will certify that a mapping or reference file is valid, for some limited definition of *valid*. By design only valid files can be submitted to the CRDS server and archive.

4.4.1 Certifying Mappings

For Mappings, crds.certify will ensure that:

- the mapping and it's descendents successfully load
- the mapping checksum is valid
- the mapping does not contain hostile code
- the mapping defines certain generic parameters
- references required by the mapping exist on the local file system

You can check the validity of your mapping or reference file like this:

```
% python -m crds.certify /where/it/really/is/hst_acs_my_masterpiece.rmap
0 errors
0 warnings
0 infos
```

By default, running certify on a mapping *does not* verify that the required reference files are valid, only that they exist.

Later versions of CRDS may have additional semantic checks on the correctness of Mappings but these are not yet implemented and hence fall to the developer to verify in some other fashion.

4.4.2 Certifying Reference Files

For reference files certify has better semantic checks. For reference files, crds.certify currently ensures that:

- the FITS format is valid
- critical reference file header parameters have acceptable values

You can certify reference files the same way as mappings, like this:

```
% python -m crds.certify /where/it/is/my_reference_file.fits
0 errors
0 warnings
0 infos
```

4.5 Mapping Checksums

CRDS mappings contain sha1sum checksums over the entire contents of the mapping, with the exception of the checksum itself. When a CRDS Mapping of any kind is loaded, the checksum is transparently verified to ensure that the Mapping contents are intact.

4.5.1 Ignoring Checksums!

Ordinarily, during pipeline operations, ignoring checksums should not be done. Ironically though, the first thing you may want to do as a developer is ignore the checksum while you load a mapping you've edited:

```
>>> pipeline = rmap.load_mapping("hst.pmap", ignore_checksum=True)
```

4.5.2 Adding Checksums

Once you've finished your masterpiece ReferenceMapping, it can be sealed with a checksum like this:

```
% python -m crds.checksum /where/it/really/is/hst_acs_my_masterpiece.rmap
```

4.6 Which Mappings Use this File?

Particularly in legacy contexts, such as HST, reference file names can be rather cryptic. Further, by design CRDS will have a complex set of fluid and versioned mappings. Hence it may become rather difficult for a human to discern which mappings refer to a particular mapping or reference file. CRDS has the crds.uses module to help answer this question:

```
% python -m crds.uses hst kcb1734ij_a2d.fits
hst.pmap
hst_acs.imap
hst_acs_atodtab.rmap
```

The first parameter indicates the observatory for which files should be considered. Additional parameters specify mapping or reference files which are used. The printed result consists of those mappings which directly or indirectly refer to the used files.

Note that the above results represent the highly simplified context of the current HST prototype, prior to the introduction of mapping evolution and version numbering. In practice, each of the above files might include several numbered versions, and some versions of the above files might not require kcb1734ij_a2d.fits.

crds.uses knows about only the mappings cached locally. Hence the official CRDS server will have a more definitive answer than someone's development machine. The CRDS web site has a link for running crds.uses over all known "official" mappings. crds.uses is especially applicable for understanding the implications of blacklisting a particular file; when a file is blacklisted, all files indicated by crds.uses are also blacklisted.

4.7 Finding Matches for a Reference in a Context

Given a particular context and reference file name, CRDS can also determine all possible matches for the reference within that context:

```
% python -m crds.matches hst.pmap kcb1734ij_a2d.fits

((('observatory', 'hst'), ('INSTRUME', 'acs'), ('filekind', 'atodtab')), (('DETECTOR', 'HRC'), ('CCDC
((('observatory', 'hst'), ('INSTRUME', 'acs'), ('filekind', 'atodtab')), (('DETECTOR', 'HRC'), ('CCDC
((('observatory', 'hst'), ('INSTRUME', 'acs'), ('filekind', 'atodtab')), (('DETECTOR', 'HRC'), ('CCDC
((('observatory', 'hst'), ('INSTRUME', 'acs'), ('filekind', 'atodtab')), (('DETECTOR', 'HRC'), ('CCDC
((('observatory', 'hst'), ('INSTRUME', 'acs'), ('filekind', 'atodtab')), (('DETECTOR', 'HRC'), ('CCDC
((('observatory', 'hst'), ('INSTRUME', 'acs'), ('filekind', 'atodtab')), (('DETECTOR', 'HRC'), ('CCDC
((('observatory', 'hst'), ('INSTRUME', 'acs'), ('filekind', 'atodtab')), (('DETECTOR', 'HRC'), ('CCDC
((('observatory', 'hst'), ('INSTRUME', 'acs'), ('filekind', 'atodtab')), (('DETECTOR', 'HRC'), ('CCDC
((('observatory', 'hst'), ('INSTRUME', 'acs'), ('filekind', 'atodtab')), (('DETECTOR', 'HRC'), ('CCDC
((('observatory', 'hst'), ('INSTRUME', 'acs'), ('filekind', 'atodtab')), (('DETECTOR', 'HRC'), ('CCDC
((('observatory', 'hst'), ('INSTRUME', 'acs'), ('filekind', 'atodtab')), (('DETECTOR', 'HRC'), ('CCDC
((('observatory', 'hst'), ('INSTRUME', 'acs'), ('filekind', 'atodtab')), (('DETECTOR', 'HRC'), ('CCDC
((('observatory', 'hst'), ('INSTRUME', 'acs'), ('filekind', 'atodtab')), (('DETECTOR', 'HRC'), ('CCDC
((('observatory', 'hst'), ('INSTRUME', 'acs'), ('filekind', 'atodtab')), (('DETECTOR', 'HRC'), ('CCDC
...
```

What is printed out is a sequence of match tuples, with each tuple nominally consisting of three parts:

```
(pmap_imap_rmap_path, match, use_after)
```

Each part in turn consists of nested tuples of the form:

```
(parkey, value)
```

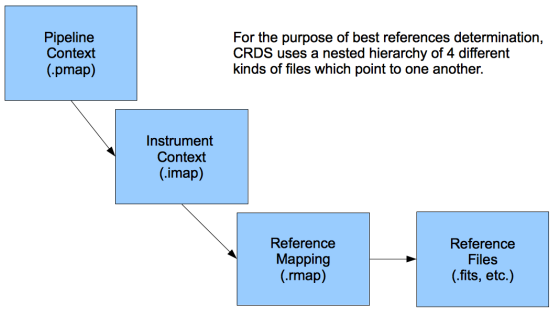
CHAPTER

FIVE

ABOUT MAPPINGS

CRDS mappings are organized in a 3 tier hierarchy: pipeline (.pmap), instrument (.imap), and reference (.rmap). Based on dataset parameters, the pipeline context is used to select an instrument mapping, the instrument mapping is used to select a reference mapping, and finally the reference mapping is used to select a reference file.

CRDS mappings are written in a subset of Python and given the proper global definitions can be parsed directly by the Python interpreter. Nothing precludes writing a parser for CRDS mappings in some other language.



5.1 Naming

The CRDS HST mapping prototypes which are generated from information scraped from the CDBS web site are named with the forms:

```
<observatory> .pmap                                 .e.g. hst.pmap
<observatory> _ <instrument> .imap                  .e.g. hst_acs.imap
<observatory> _ <instrument> _ <filekind> .rmap     .e.g. hst_acs_darkfile.rmap
```

The names of subsequent derived mappings include a version number:

```
<observatory> _ <version> .pmap                                  .e.g. hst_00001.pmap
<observatory> _ <instrument> _ <version> .imap                   .e.g. hst_acs_00047.imap
<observatory> _ <instrument> _ <filekind> _ <version> .rmap  .e.g. hst_acs_darkfile_00012.rmap
```

5.2 Basic Structure

All mappings have the same basic structure consisting of a "header" section followed by a "selector" section. The header provides meta data describing the mapping, while the selector provides matching rules used to look up the results of the mapping. A critical field in the mapping header is the "parkey" field which names the dataset header parameters which are used by the selector to do its lookup.

5.3 Pipeline Mappings (.pmap)

A sample pipeline mapping for HST looks like:

```
header = {
    'name' : 'hst.pmap',
    'derived_from' : 'created by hand 12-23-2011',
    'mapping' : 'PIPELINE',
    'observatory' : 'HST',
    'parkey' : ('INSTRUME',),
    'description' : 'Initially generated on 12-23-2011',
    'sha1sum' : 'e2c6392fd2731df1e8d933bd990f3fd313a813db',
}

selector = {
    'ACS' : 'hst_acs.imap',
    'COS' : 'hst_cos.imap',
    'NICMOS' : 'hst_nicmos.imap',
    'STIS' : 'hst_stis.imap',
    'WFC3' : 'hst_wfc3.imap',
    'WFPC2' : 'hst_wfpc2.imap',
}
```

A pipeline mapping matches the dataset "INSTRUME" header keyword against its selector to look up an instrument mapping file.

5.4 Instrument Mappings (.imap)

A sample instrument mapping for HST's COS instrument looks like:

```
header = {
    'derived_from' : 'scraped 2011-12-23 11:57:10',
    'description' : 'Initially generated on 2011-12-23 11:57:10',
    'instrument' : 'COS',
    'mapping' : 'INSTRUMENT',
    'name' : 'hst_cos.imap',
    'observatory' : 'HST',
    'parkey' : ('REFTYPE',),
    'sha1sum' : '800fb1567cb5bed4031402c7396aeb86c5e1db61',
    'source_url' : 'http://www.stsci.edu/hst/observatory/cdbs/SIfileInfo/COS/reftablequeryindex',
}

selector = {
    'badttab' : 'hst_cos_badttab.rmap',
    'bpixtab' : 'hst_cos_bpixtab.rmap',
    'brftab' : 'hst_cos_brftab.rmap',
    'brsttab' : 'hst_cos_brsttab.rmap',
```

```
    'deadtab' : 'hst_cos_deadtab.rmap',
    'disptab' : 'hst_cos_disptab.rmap',
    'flatfile' : 'hst_cos_flatfile.rmap',
    'fluxtab' : 'hst_cos_fluxtab.rmap',
    'geofile' : 'hst_cos_geofile.rmap',
    'lamptab' : 'hst_cos_lamptab.rmap',
    'phatab' : 'hst_cos_phatab.rmap',
    'spwcstab' : 'hst_cos_spwcstab.rmap',
    'tdstab' : 'hst_cos_tdstab.rmap',
    'wcptab' : 'hst_cos_wcptab.rmap',
    'xtractab' : 'hst_cos_xtractab.rmap',
}
```

Instrument mappings match the desired reference file type against the reference mapping which can be used to determine a best reference recommendation for a particular dataset. An instrument mapping lists all possible reference types for all modes of the instrument, some of which may not be appropriate for a particular mode. The selector key of an instrument mapping is the value of a reference file header keyword "REFTYPE", and is the name of the dataset header keyword which will record the best reference selection.

5.5 Reference Mappings (.rmap)

A sample reference mapping for HST COS DEADTAB looks like:

```
header = {
    'derived_from' : 'scraped 2011-12-23 11:54:56',
    'description' : 'Initially generated on 2011-12-23 11:54:56',
    'filekind' : 'DEADTAB',
    'instrument' : 'COS',
    'mapping' : 'REFERENCE',
    'name' : 'hst_cos_deadtab.rmap',
    'observatory' : 'HST',
    'parkey' : (('DETECTOR',), ('DATE-OBS', 'TIME-OBS')),
    'sha1sum' : 'e27984a6441d8aaa7cd28ead2267a6be4c3a153b',
}

selector = Match({
    ('FUV',) : UseAfter({
        '1996-10-01 00:00:00' : 's7g1700gl_dead.fits',
    }),
    ('NUV',) : UseAfter({
        '1996-10-01 00:00:00' : 's7g1700ql_dead.fits',
    }),
})
```

Reference mapping selectors are constructed as a nested hierarchy of selection operators which match against various dataset header keywords.

5.5.1 Parkeys

For reference mappings, the header "parkey" field is a tuple of tuples. Each stage of the nested selector consumes the next tuple of header keys. For the example above, the Match operator matches against the value of the dataset keyword "DETECTOR". Based on that match, the selected UseAfter operator matches against the dataset's "DATE-OBS" and "TIME-OBS" keywords to lookup the name of a reference file.

5.6 Selectors

All the CRDS selection operators are written to select either a filename *or* a nested operator. In the case of HST, the Match operator locates a nested UseAfter operator which in turn locates the reference file.

5.6.1 Match

Based on a dataset's header values, Match locates the match tuple which best matches the dataset. Conceptually this is a dictionary lookup. In actuality, CRDS processes each match parameter in succession, at each step eliminating match candidates that cannot possibly match.

Parameter Tuples and Simple Matches

The CRDS Match operator typically matches a dataset header against a tuple which defines multiple parameter values whose names are specified in the rmap header `parkey`:

```
("UVIS", "F122LP")    :  'some_file_or_nested_selection'
```

Alternately, for simple use cases the Match operator can match against single strings, which is a simplified syntax for a 1-tuple:

```
'UVIS'  :  'some_file_or_nested_selection'
('UVIS',) : 'this_is_the_equivalent_one_tuple'
```

Single Parameter Values

Each value within the match tuples of a Match operator can be an expression in its own right. There are a number of special values associated with each match expression: Ors |, Wildcards *, Regular Expressions (), Literals {}, Relationals, between, N/A, and Substitutions.

Or |

Many CRDS match expressions consist of a series of match patterns separated by vertical bars. The vertical bar is read as "or" and means that a match occurs if either pattern matches that dataset header. For example, the expression:

```
("either_this|that","1|2|3")  : "some_file.fits"
```

will match:

```
("either_this", "2")
```

and also:

```
("that", "1")
```

Wild Cards *

By default, * is interpreted in CRDS as a glob pattern, much like UNIX shell file name matching. * matches any sequence of characters. The expression:

```
("F*122",) : "some_file.fits"
```

will match any value starting with "F" and ending with "122".

Regular Expressions

CRDS can match on true regular expressions. A true regular expression match is triggered by bracketing the match in parentheses ():

```
("(^F[^13]22$)",)  : "some_file.fits"
```

The above corresponds to matching the regular expression "^F[^1234]22$" (note that the bracketing parentheses within the string are removed.) Regular expression syntax is explained in the Python documentation for the re module. The above expression will match values starting with "F", followed by any character which is not "1" or "3" followed by "22".

Literal Expressions

A literal expression is bracketed with curly braces {} and is matched without any interpretation whatsoever. Hence, special characters like * or | are interpreted literally rather than as ors or wildcards. The expression:

```
("{F|*G}",) : "some_file.fits"
```

matches the value "F|*G" as opposed to "F" or anything ending with "G".

Relational Expressions

Relational expressions are bracketed by the pound character #. Relational expressions do numerical comparisons on the header value to determine a match. Relational expressions have implicit variables and support the operators:

```
> >= < <= == and or
```

The expression:

```
("# >1 and <37 #",)  : "some_file.fits"
```

will match any number greater than 1 and less than 37.

Between

A special relational operator "between" is used to simply express a range of numbers >= to the lower bound and < the upper bound, similar to Python slicing:

```
("between 1  47",) : "some_file.fits"
```

will match any number greater than or equal to 1 and less than 47. This is equivalent to:

```
("# >=1 and <47 #",) : "some_file.fits"
```

Note that "between" matches sensibly stack into a complete range. The expressions:

```
("between 1 47",) : "some_file.fits"
("between 47 90", ) : "another_file.fits"
```

provide complete coverage for the range between 1 and 90.

N/A

Some rmaps have match tuple values of "N/A", or Not Applicable. A value of N/A is matched as a special version of "*", matching anything, but not affecting the "weight" of the match.

('HRC', 'N/A') : "some_file.fits"

There are a couple uses for N/A parameters. First, sometimes a parameter is irrelevant in the context of the other parameters. So for an rmap which covers multiple instrument modes, a parameter may not apply to all modes. Second, sometimes a parameter is relevant to custom lookup code, but is not used by the match directly. In this second case, the "N/A" parameter may be used by custom header preconditioning code to assist in mutating the other parameter values that *are* used in the match.

Substitution Parameters

Substituion parameters are short hand notation which eliminate the need to duplicate rmap rules. In order to support WFC3 biasfile conventions, CRDS rmaps permit the definition of meta-match-values which correspond to a set of actual dataset header values. For instance, when an rmap header contains a "substitutions" field like this:

```
'substitutions' : {
    'CCDAMP' : {
        'G280_AMPS' : ('ABCD', 'A', 'B', 'C', 'D', 'AC', 'AD', 'BC', 'BD'),
    },
},
```

then a match tuple line like the following could be written:

```
('UVIS', 'G280_AMPS', '1.5', '1.0', '1.0', 'G280-REF', 'T') : UseAfter({
```

Here the value of G280_AMPS works like this: first, reference files listed under that match tuple define CCDAMP=G280_AMPS. Second, datasets which should use those references define CCDAMP to a particular amplifier configuration, .e.g. ABCD. Hence, the reference file specifies a set of applicable amplifier configurations, while the dataset specifies a particular configuration. CRDS automatically expands substitutions into equivalent sets of match rules.

Match Weighting

Because of the presence of special values like regular expressions, CRDS uses a winnowing match algorithm which works on a parameter-by-parameter basis by discarding match tuples which cannot possibly match. After examining all parameters, CRDS is left with a list of candidate matches.

For each literal, *, or regular expression parameter that matched, CRDS increases its sense of the goodness of the match by 1. For each N/A that was ignored, CRDS doesn't change the weight of the match. The highest ranked match is the one CRDS chooses as best. When more than one match tuple has the same highest rank, we call this an "ambiguous" match. Ambiguous matches will either be merged, or treated as errors/exceptions that cause the match to fail. Talk about ambiguity.

For the initial HST rmaps, there are a number of match cases which overlap, creating the potential for ambiguous matches by actual datasets. For HST, all of the match cases refer to nested UseAfter selectors. A working approach for handling ambiguities here is to merge the two or more equal weighted UseAfter lists into a single combined UseAfter which is then searched.

The ultimate goal of CRDS is to produce clear non-overlapping rules. However, since the initial rmaps are generated from historical mission data in CDBS, there are eccentricities which need to be accomodated by merging or eventually addressed by human beings who will simplify the rules by hand.

5.6.2 UseAfter

The UseAfter selector matches an ordered sequence of date time values to corresponding reference filenames. UseAfter finds the greatest date-time which is less than or equal to (<=) EXPSTART of a dataset. Unlike reference file and dataset timestamp values, all CRDS rmaps represent times in the single format shown in the rmap example below:

```
selector = Match({
    ('HRC',) : UseAfter({
        '1991-01-01 00:00:00' : 'j4d1435hj_a2d.fits',
        '1992-01-01 00:00:00' : 'kcb1734ij_a2d.fits',
    }),
    ('WFC',) : UseAfter({
        '1991-01-01 00:00:00' : 'kcb1734hj_a2d.fits',
        '2008-01-01 00:00:00' : 't3n1116mj_a2d.fits',
    }),
})
```

In the above mapping, when the detector is HRC, if the dataset's date/time is before 1991-01-01, there is no match. If the date/time is between 1991-01-01 and 1992-01-01, the reference file 'j4d1435hj_a2d.fits' is matched. If the dataset date/time is 1992-01-01 or after, the recommended reference file is 'kcb1734ij_a2d.fits'

5.6.3 SelectVersion

The SelectVersion() rmap operator uses a software version and various relations to make a selection:

```
selector = SelectVersion({
    '<3.1':    'cref_flatfield_65.fits',
    '<5':      'cref_flatfield_73.fits',
    'default': 'cref_flatfield_123.fits',
})
```

While similar to relational expressions in Match(), SelectVersion() is dedicated, simpler, and more self-documenting. With the exception of default, versions are examined in sorted order.

5.6.4 ClosestTime

The ClosestTime() rmap operator does a lookup on a series of times and selects the closest time which either precedes or follows the given parameter value:

```
selector = ClosestTime({
     '2017-04-24 00:00:00':  "cref_flatfield_123.fits",
     '2018-02-01 00:00:00' : "cref_flatfield_222.fits",
     '2019-04-15 00:00:00':  "cref_flatfield_123.fits",
})
```

So a parameter of '2017-04-25 00:00:00' would select 'cref_flatfield_123.fits'.

5.6.5 GeometricallyNearest

The GeometricallyNearest() selector applies a distance relation between a numerical parameter and the match values. The match value which is closest to the supplied parameter is chosen:

```
selector = GeomtricallyNearest({
    1.2 : "cref_flatfield_120.fits",
    1.5 : "cref_flatfield_124.fits",
```

```
    5.0 : "cref_flatfield_137.fits",
})
```

In this case, a value of 1.3 would match 'cref_flatfield_120.fits'.

5.6.6 Bracket

The Bracket() selector is unusual because it returns the pair of selections which enclose the supplied parameter value:

```
selector = Bracket({
    1.2: "cref_flatfield_120.fits",
    1.5: "cref_flatfield_124.fits",
    5.0: "cref_flatfield_137.fits",
})
```

Here, a parameter value of 1.3 returns the value:

```
('cref_flatfield_120.fits', 'cref_flatfield_124.fits')
```