



crds Documentation

Release 1.1

STScI

June 03, 2015

1	Package Overview	1
2	Installation	3
2.1	Installation via Ureka	3
2.2	Installing from Source	3
2.3	Dependencies	3
3	Setting up your Environment	5
3.1	Basic Environment	5
3.2	Setup for On Site Operational Use (HST or JWST)	5
3.3	Setup for Offsite Use	7
3.4	Advanced Environment	8
4	Command Line Tools	11
4.1	Specifying File Paths	11
4.2	crds.bestrefs	11
4.3	crds.sync	15
4.4	crds.certify	18
4.5	crds.diff	20
4.6	crds.rowdiff	21
4.7	crds.uses	22
4.8	crds.matches	23
4.9	pipeline_bestrefs	24
5	Library Use	25
5.1	crds.getreferences()	25
5.2	crds.get_default_context()	26
6	Core Library Functions	27
6.1	Overview of Features	27
6.2	Important Modules	27
6.3	Basic Operations on Mappings	28
6.4	Mapping Checksums	30
7	About Mappings	33
7.1	Naming	33
7.2	Basic Structure	34
7.3	Pipeline Mappings (.pmap)	34
7.4	Instrument Mappings (.imap)	34
7.5	Reference Mappings (.rmap)	35
7.6	Active Header Fields	36

7.7	Selectors	39
8	Using the CRDS Web Site	45
8.1	Operational References	45
8.2	Context History (more)	46
8.3	Open Services	47
8.4	Private Functions	52
9	Web Services	61
9.1	Supported Methods	61
9.2	JSONRPC URL	62
9.3	JSONRPC Request	62
9.4	JSONRPC Response	62
9.5	Error Handling	63
9.6	JSONRPC Demo Page	63
10	CRDS Database Access	65
10.1	JSON RPC Access	65
10.2	Download CRDS catalog for SQLite queries	66

PACKAGE OVERVIEW

The CRDS client and command line software is distributed as a single package with several sub-packages:

- **crds**
 - core package enabling local use and development of mappings and reference files. contains command line utility programs.
- **crds.cache**
 - prototype cache which contains the original baseline CRDS mappings generated for HST and JWST, also demonstrating cache structure for a dual project cache.
- **crds.client**
 - network client library for interacting with the central CRDS server. This is primarily for internal use in CRDS, encapsulating JSONRPC interfaces with Python.
- **crds.hst**
 - observatory personality package for HST, defining how HST types, reference file certification constraints, and naming works.
- **crds.jwst**
 - analogous to crds.hst, for JWST.
- **crds.tobs**
 - test observatory supporting artificial rules cases and tests.

CRDS also contains a number of command line tools:

- **crds.bestrefs**
 - Best references utility for HST FITS files and context-to-context affected datasets computations.
- **crds.sync**
 - Cache download and maintenance tool, fetches, removes, checks, and repairs rules and references.

- **crds.certify**
 - Checks constraints and format for CRDS rules and references.
- **crds.diff, crds.rowdiff**
 - Difference utility for rules and references, also FITS table differences.
- **crds.matches**
 - Prints out parameter matches for particular references, or database matching parameters with respect to particular dataset IDs.
- **crds.uses**
 - Lists files which refer to (are dependent on) some CRDS rules or reference file.
- **crds.list**
 - Lists cache files and configuration, prints rules files, dumps database dataset parameter dictionaries.

More information can be found on each tool using the command line – `--help` switch, e.g.:

```
% python -m crds.bestrefs --help
```

or in the command line tools section of this document.

INSTALLATION

2.1 Installation via Ureka

Ureka is a collection of astronomy software that provides everything you need to run the data reduction packages provided by STScI. Most people install CRDS as part of Ureka which is found here:

<http://ssb.stsci.edu/ureka/>

Follow the instructions for installing Ureka and afterward you should be able to do the following:

```
% python -m crds.list --version CRDS : INFO crds version 1.1.1 revision 1784M
```

and get similar output reflecting your installed CRDS version.

2.2 Installing from Source

2.2.1 Subversion Checkout

Alternately, CRDS source code can be downloaded from the CRDS subversion repository like this:

```
% svn co https://aeon.stsci.edu/ssb/svn/crds/trunk crds
```

2.2.2 Run the Install Script

Installing from source, run the install script in the root source code directory:

```
% cd crds
% ./install
final status 000000
```

2.3 Dependencies

CRDS was developed in and for an STSCI Python environment suitable for pipeline processing. Standard STScI calibration environments should already include it. Nevertheless, for installing CRDS independently, these dependencies are applicable:

REQUIRED: CRDS requires these dependencies to be installed in your Python environment:

- numpy
- astropy

OPTIONAL: For executing the unit tests (runtests) add:

- nose
- BeautifulSoup
- stsci.tools

OPTIONAL: For running `crds.certify` to fully check CRDS rules/mapping files add:

- Parsley-1.1 (included in CRDS subversion under `third_party`)
- pyaml (for certifying and using yaml references)
- pyasdf (for certifying and using ASDF references)

OPTIONAL: For building documentation add:

- docutils
- sphinx
- stsci.sphinxext

SETTING UP YOUR ENVIRONMENT

CRDS is used in a number of different contexts and consequently is configurable. The defaults for CRDS are tuned for onsite use at STScI using operational references, requiring little or no configuration onsite. Subsequent instructions are provided for setting up more personalized or offsite environments.

3.1 Basic Environment

CRDS supports HST and JWST projects using project-specific servers and an explicit cache of CRDS rules and reference files. CRDS has two environment variables which define basic setup. These variables control the server where CRDS obtains rules and references and where CRDS caches files to on your local system:

```
% setenv CRDS_SERVER_URL    <some_crds_server>
% setenv CRDS_PATH          <some_crds_reference_and_rules_cache_directory>
```

If you are currently working on only a single project, it may be helpful to declare that project:

```
% setenv CRDS_OBSERVATORY   hst (or jwst)
```

3.2 Setup for On Site Operational Use (HST or JWST)

This section describes use of operational reference files onsite at STScI. It's relevant to fully archived operational files, not development and test.

3.2.1 File Cache Location (CRDS_PATH)

For typical onsite use at STScI, CRDS users can share a file cache which contains all rules and references. The location of the shared cache initially defaults to:

```
/grp/crds/cache
```

/grp/crds/cache is designed to support both HST and JWST with a single defaulted **CRDS_PATH** setting.

Since /grp/crds/cache is the default, you don't have to explicitly set **CRDS_PATH**.

Since /grp/crds/cache starts out containing all the operational CRDS rules and reference files, file downloads are not required.

3.2.2 Server Selection (CRDS_SERVER_URL)

Since each project is supported by a different operational server, CRDS must determine which (if any) server to use. Starting with OPUS 2014.3 and crds-1.1, CRDS does a reasonable job guessing what project you're working on.

CRDS can guess the project you're working on by:

- Looking for the string 'hst' or 'jwst' in the file names you're operating on.
- Looking inside files to determine the applicable instrument, and inferring the project from the instrument name.
- If you explicitly set CRDS_SERVER_URL, CRDS can ask the server which project it supports.

You can tell CRDS which project you're working on by:

- Using command line switches in CRDS utility programs: `—hst` or `—jwst`
- Setting CRDS_OBSERVATORY to 'hst' or 'jwst'

If you're working on both projects frequently, using the command line hints, e.g. `—hst`, is probably preferred whenever CRDS has trouble guessing.

If you're primarily working on one project, defining **CRDS_OBSERVATORY** is probably most convenient since then you won't need to provide command line hints.

If CRDS can determine the project, and you don't specify CRDS_SERVER_URL, CRDS will use the default operational server for your project:

Project	Implicit CRDS_SERVER_URL
hst	https://hst-crds.stsci.edu
jwst	https://jwst-crds.stsci.edu

If CRDS cannot determine your project, and you did not specify CRDS_SERVER_URL, it will be defaulted to:

<https://crds-serverless-mode.stsci.edu>

In serverless mode, dynamic cache updates are not possible so cache information may become stale. This affects CRDS rules and reference updates, CRDS knowledge of the current operational context, and CRDS knowledge of rules or references determined to be bad. On the other hand, in serverless-mode you're guaranteed to be working with a static system, and no warnings will be issued because the server is not reachable.

3.2.3 Onsite CRDS Testing

For reference type development, updates are generally made and tested in the test pipelines at STScI. For coordinating with those tests, **CRDS_PATH** and **CRDS_SERVER_URL** must be explicitly set to a test cache and server similar to this:

```
% setenv CRDS_PATH ${HOME}/crds_cache_test
% setenv CRDS_SERVER_URL https://hst-crds-test.stsci.edu
```

After syncing this will provide access to CRDS test files and rules in a local cache:

```
# Fetch all the test rules
% python -m crds.sync --all
# Fetch specifically listed test references
% python -m crds.sync --files <test_references_only_the_test_server_has...>
```

Testing reference type changes (new keywords, new values or value restrictions, etc) may also require access to development versions of CRDS code. In particular, when adding parameters or changing legal parameter values, the certify tool is modified as “code” on the servers first. Hence distributed versions of CRDS will not reflect ongoing type changes. The test server Certify Files function should generally reflect the most up-to-date knowledge CRDS has

about ongoing type changes. To see how new reference files stack up with changing CRDS code, try submitting the files to Certify Files on the test server or ask what the status is on crds_team@stsci.edu.

NOTE: the test server is only visible on-site, not on the internet. Without VPN or port forwarding, the test servers are not usable off site.

3.3 Setup for Offsite Use

CRDS has been designed to (optionally) automatically fetch and cache references you need to process your datasets. Rather than going to a website and downloading a tarball of recommended references, the CRDS tools, which know the references you need, can go to the website for you and download the files you need to your cache. Once you've cached a file, unless you delete it, you never have to download it again.

For offsite users without VPN access who are running local calibrations, you can create a small personal cache of rules and references supporting only the datasets you care about:

```
% setenv CRDS_PATH ${HOME}/crds_cache
```

For **HST**, to fetch the references required to process some FITS datasets:

```
% python -m crds.bestrefs --files dataset*.fits --sync-references=1
```

By default `crds.bestrefs` does not alter your dataset FITS files. If you also wish to update your dataset FITS headers with best references, add `-update-bestrefs`.

For **JWST**, CRDS is directly integrated with the calibration step code and will automatically download rules and references as needed. Downloads will only be an issue when you set `CRDS_PATH` and don't already have the files you need in your cache. By default CRDS modifies JWST datasets with new best references which serve as a processing history in the dataset header.

Users of `/grp/crds/cache` cannot update the readonly cache so they should not attempt to run `crds.sync` or fetch references with `crds.bestrefs`. `/grp/crds/cache` should always be complete within a few hours of archiving any new reference or rules delivery, changing the operational context, or marking files bad.

3.3.1 Additional HST Settings

HST calibration steps access reference files indirectly through environment variables. There are two forms of CRDS cache reference file organization: flat and with instrument subdirectories. The original CRDS cache format was flat, and the shared group cache at `/grp/crds/cache` remains flat.

Flat CRDS cache For calibration software to use references in a CRDS cache with a flat reference file organization, including the default shared group readonly cache at `/grp/crds/cache`, set these environment variables:

```
setenv iref ${CRDS_PATH}/references/hst/
setenv jref ${CRDS_PATH}/references/hst/
setenv oref ${CRDS_PATH}/references/hst/
setenv lref ${CRDS_PATH}/references/hst/
setenv nref ${CRDS_PATH}/references/hst/
setenv uref ${CRDS_PATH}/references/hst/
setenv uref_linux $$uref
```

By-Instrument CRDS cache For calibration software to use references in a CRDS cache with a by-instrument organization, the default for newly created caches in the future, set these environment variables:

```
setenv iref ${CRDS_PATH}/references/hst/iref/
setenv jref ${CRDS_PATH}/references/hst/jref/
setenv oref ${CRDS_PATH}/references/hst/oref/
```

```
setenv lref ${CRDS_PATH}/references/hst/lref/
setenv nref ${CRDS_PATH}/references/hst/nref/
setenv uref ${CRDS_PATH}/references/hst/uref/
setenv uref_linux $uref
```

Reorganizing CRDS References The `crds.sync` tool can be used to reorganize the directory structure of a large existing CRDS cache as follows to switch from flat to by-instrument:

```
python -m crds.sync --organize=instrument

# or to switch from by-instrument to flat

python -m crds.sync --organize=flat
```

Another simpler approach is to delete and recreate your existing cache, more feasible for small personal caches than for complete terabyte-scale caches.

3.3.2 JWST Context

The CRDS context used to evaluate CRDS best references for JWST defaults to `jwst-operational`, the changing symbolic context which is in use in the JWST pipeline. During early development `jwst-operational` corresponds to the latest context which is sufficiently mature for broad use. Use of `jwst-operational` is automatic.

The context used for JWST can be overridden to some specific historical or experimental context by setting the **CRDS_CONTEXT** environment variable:

```
% setenv CRDS_CONTEXT jwst_0057.pmap
```

CRDS_CONTEXT does not override command line switches or parameters passed explicitly to `crds.getreferences()`.

3.4 Advanced Environment

A number of things in CRDS are configurable with environment variables, most important of which is the location and structure of the file cache.

3.4.1 Multi-Project Caches

CRDS_PATH defines a cache structure for multiple projects. Each major branch of a multi-project cache contains project specific subdirectories:

```
/cache
  /mappings
    /hst
      hst mapping files...
    /jwst
      jwst mapping files...
  /references
    /hst
      hst reference files...
    /jwst
      jwst reference files...
  /config
    /hst
      hst config files...
```

```

/jwst
  jwst config files...

```

- *mappings* contains versioned rules files for CRDS reference file assignments
- *references* contains reference files themselves
- *config* contains system configuration information like operational context and bad files

Individual branches of a cache can be overridden to locate that branch outside the directory tree specified by `CRDS_PATH`. The remaining directories can be overridden as well or derived from `CRDS_PATH`.

CRDS_MAPPATH can be used to override `CRDS_PATH` and define where only mapping files are stored. `CRDS_MAPPATH` defaults to `${CRDS_PATH}/mappings` which contains multiple observatory-specific subdirectories.

CRDS_REFPATH can be used to override `CRDS_PATH` and define where only reference files are stored. `CRDS_REFPATH` defaults to `${CRDS_PATH}/references` which contains multiple observatory specific subdirectories.

CRDS_CFGPATH can be used to override `CRDS_PATH` and define where only configuration information is cached. `CRDS_CFGPATH` defaults to `${CRDS_PATH}/config` which can contain multiple observatory-specific subdirectories.

Specifying `CRDS_MAPPATH = /somewhere` when `CRDS_OBSERVATORY = hst` means that mapping files will be located in `/somewhere/hst`.

While it can be done, it's generally considered an error to use a multi-project cache with different servers for the *same* observatory, e.g. both `hst-test` and `hst-ops`.

3.4.2 Single Project Caches

CRDS_PATH_SINGLE defines a cache structure for a single project. The component paths implied by **CRDS_PATH_SINGLE** omit the observatory subdirectory, giving a simpler and shallower cache structure:

```

/cache
  /mappings
    mapping_files...
  /references
    reference_files...
  /config
    config_files...

```

It's an error to use a single project cache with more than one project or server. It is inadvisable to mix multi-project (no `_SINGLE`) and single-project (`_SINGLE`) configuration variables, set one or the other form, not both.

As with **CRDS_PATH**, there are overrides for each cache branch which can locate it independently.

CRDS_MAPPATH_SINGLE can be used to override `CRDS_PATH` and define where only mapping files are stored. `CRDS_MAPPATH_SINGLE` defaults to `${CRDS_PATH}/mappings` but is presumed to support only one observatory.

CRDS_REFPATH_SINGLE can be used to override `CRDS_PATH` and define where only reference files are stored. `CRDS_REFPATH_SINGLE` defaults to `${CRDS_PATH}/references` but is presumed to support only one observatory.

CRDS_CFGPATH_SINGLE can be used to override `CRDS_PATH` and define where only server configuration information is cached. `CRDS_CFGPATH_SINGLE` defaults to `${CRDS_PATH}/config` but is presumed to support only one observatory.

Specifying `CRDS_MAPPATH_SINGLE = /somewhere` when `CRDS_OBSERVATORY = hst` means that mapping files will be located in `/somewhere`, not in `/somewhere/hst`.

3.4.3 Miscellaneous Variables

CRDS_VERBOSITY enables output of CRDS debug messages. Set to an integer, nominally 50. Higher values output more information, lower values less information. CRDS also has command line switches `-verbose` (level=50) and `-verbosity=<level>`. Verbosity level ranges from 0 to 100 and defaults to 0 (no verbose output).

CRDS_ALLOW_BAD_RULES enable CRDS to use assignment rules which have been designated as bad files / scientifically invalid.

CRDS_ALLOW_BAD_REFERENCES enable CRDS to assign reference files which have been designated as scientifically invalid after issuing a warning.

CRDS_IGNORE_MAPPING_CHECKSUM causes CRDS to waive mapping checksums when set to True, useful when you're editing them.

CRDS_READONLY_CACHE limits tools to readonly access to the cache when set to True. Eliminates cache writes which occur implicitly. This is mostly useful in CRDS server user cases which want to ensure not modifying the server CRDS cache but cannot write protect it effectively.

CRDS_MODE defines whether CRDS should compute best references using installed client software only (local), on the server (remote), or intelligently “fall up” to the server (when the installed client is deemed obsolete relative to the server) or “fall down” to the local installation (when the server cannot be reached) (auto). The default is auto.

CRDS_CLIENT_RETRY_COUNT number of times CRDS will attempt a network transaction with the CRDS server. Defaults to 1 meaning 1 try with no retries.

CRDS_CLIENT_RETRY_DELAY_SECONDS number of seconds CRDS waits after a failed network transaction before trying again. Defaults to 0 seconds, meaning proceed immediately after fail.

COMMAND LINE TOOLS

Using the command line tools requires a local installation of the CRDS library. Some of the command line tools also interact with the CRDS server in order to implement their functionality.

4.1 Specifying File Paths

The command line tools operate on CRDS reference and mapping files in various ways. To specify a file in your local CRDS file cache, as defined by `CRDS_PATH`, use no path on the file:

```
% python -m crds.diff hst.pmap hst_0001.pmap # assumes paths in CRDS cache
```

To specify a particular file which is not located in your cache, give at least a relative path to the file, `./` will do:

```
% python -m crds.diff /some/path/hst.pmap ./hst_0002.pmap # uses given paths
```

4.2 `crds.bestrefs`

`crds.bestrefs` computes the best references with respect to a particular context or contexts for a set of FITS files, dataset ids, or instruments:

```
usage: /Users/jmiller/virtenv/ssbdev/lib/python2.7/site-packages/crds/bestrefs.py
  [-h] [-n NEW_CONTEXT] [-o OLD_CONTEXT] [--fetch-old-headers] [-c]
  [-f FILES [FILES ...]] [-d IDS [IDS ...]] [--all-instruments]
  [-i INSTRUMENTS [INSTRUMENTS ...]]
  [-t REFERENCE_TYPES [REFERENCE_TYPES ...]]
  [-k SKIPPED_REFERENCE_TYPES [SKIPPED_REFERENCE_TYPES ...]]
  [--diffs-only] [--datasets-since DATASETS_SINCE]
  [-p [LOAD_PICKLES [LOAD_PICKLES ...]]] [-a SAVE_PICKLE]
  [--update-pickle] [--only-ids [IDS [IDS ...]]] [-u] [--print-affected]
  [--print-affected-details] [--print-new-references]
  [--print-update-counts] [-r] [-m SYNC_MAPPINGS] [-s SYNC_REFERENCES]
  [--differences-are-errors] [--allow-bad-rules] [--allow-bad-references]
  [-e] [--undefined-differences-matter] [--na-differences-matter]
  [--compare-cdbs] [--affected-datasets] [-z] [--dump-unique-errors]
  [--unique-errors-file UNIQUE_ERRORS_FILE]
  [--all-errors-file ALL_ERRORS_FILE] [-v] [--verbosity VERBOSITY] [-R]
  [-I] [-V] [-J] [-H] [--stats] [--profile PROFILE] [--log-time] [--pdb]
  [--debug-traps]
```

- * Determines best references with respect to a context or contexts.
- * Optionally compares new results to prior results.

- * Optionally prints source data names affected by the new context.
- * Optionally updates the headers of file-based data with new recommendations.

optional arguments:

```

-h, --help                show this help message and exit
-n NEW_CONTEXT, --new-context NEW_CONTEXT
                        Compute the updated best references using this context. Uses current operation
-o OLD_CONTEXT, --old-context OLD_CONTEXT
                        Compare bestrefs recommendations from two contexts.
--fetch-old-headers      Fetch old headers in accord with old parameter lists.  Slower, avoid unless
-c, --compare-source-bestrefs
                        Compare new bestrefs recommendations to recommendations from data source, fr
-f FILES [FILES ...], --files FILES [FILES ...]
                        Dataset files to compute best references for.
-d IDs [IDs ...], --datasets IDs [IDs ...]
                        Dataset ids to consult database for matching parameters and old results.
--all-instruments        Compute best references for cataloged datasets for all supported instruments
-i INSTRUMENTS [INSTRUMENTS ...], --instruments INSTRUMENTS [INSTRUMENTS ...]
                        Instruments to compute best references for, all historical datasets in databa
-t REFERENCE_TYPES [REFERENCE_TYPES ...], --types REFERENCE_TYPES [REFERENCE_TYPES ...]
                        A list of reference types to process, defaulting to all types.
-k SKIPPED_REFERENCE_TYPES [SKIPPED_REFERENCE_TYPES ...], --skip-types SKIPPED_REFERENCE_TYPES [SK
                        A list of reference types which should not be processed, defaulting to noth
--diffs-only             For context-to-context comparison, choose only instruments and types from con
--datasets-since DATASETS_SINCE
                        Cut-off date for datasets, none earlier than this. Use 'auto' to exploit re
-p [LOAD_PICKLES [LOAD_PICKLES ...]], --load-pickles [LOAD_PICKLES [LOAD_PICKLES ...]]
                        Load dataset headers and prior bestrefs from pickle files, in worst-to-best
-a SAVE_PICKLE, --save-pickle SAVE_PICKLE
                        Write out the combined dataset headers to the specified pickle file. Can al
--update-pickle          Replace source bestrefs with CRDS bestrefs in output pickle. For setting up
--only-ids [IDS [IDS ...]]
                        If specified, process only the listed dataset ids.
-u, --update-bestrefs    Update sources with new best reference recommendations.
--print-affected         Print names of products for which the new context would assign new references
--print-affected-details
                        Include instrument and affected types in addition to compound names of affect
--print-new-references   Prints one line per reference file change. If no comparison requested, prin
--print-update-counts    Prints dictionary of update counts by instrument and type, status on update
-r, --remote-bestrefs    Compute best references on CRDS server, convenience for env var CRDS_MODE='r
-m SYNC_MAPPINGS, --sync-mappings SYNC_MAPPINGS
                        Fetch the required context mappings to the local cache. Defaults TRUE.
-s SYNC_REFERENCES, --sync-references SYNC_REFERENCES
                        Fetch the references recommended by new context to the local cache. Defaults
--differences-are-errors
                        Treat recommendation differences between new context and original source as e
--allow-bad-rules        Only warn if a context which is marked 'bad' is used, otherwise error.
--allow-bad-references   Only warn if a reference which is marked bad is recommended, otherwise error
-e, --bad-files-are-errors
                        DEPRECATED / default; Recommendations of known bad/invalid files are errors,
--undefined-differences-matter
                        If not set, a transition from UNDEFINED to anything else is not considered a

```



```

--na-differences-matter      If not set, either CDBS or CRDS recommending N/A is OK to mismatch.
--compare-cdb               Abbreviation for --compare-source-bestrefs --differences-are-errors --dump-unique-errors
--affected-datasets         Abbreviation for --diffs-only --datasets-since=auto --optimize-tables --print-unique-errors
-z, --optimize-tables      If set, apply row-based optimizations to screen out inconsequential table updates.
--dump-unique-errors        Record and dump the first instance of each kind of error.
--unique-errors-file        UNIQUE_ERRORS_FILE
                           Write out data names (ids or filenames) for first instance of unique errors to specified file.
--all-errors-file           ALL_ERRORS_FILE
                           Write out all err'ing data names (ids or filenames) to specified file.
-v, --verbose               Set log verbosity to True, nominal debug level.
--verbosity VERBOSITY      Set log verbosity to a specific level: 0..100.
-R, --readonly-cache       Don't modify the CRDS cache. Not compatible with options which implicitly modify the cache.
-I, --ignore-cache         Download required files even if they're already in the cache.
-V, --version               Print the software version and exit.
-J, --jwst                 Force observatory to JWST for determining header conventions.
-H, --hst                  Force observatory to HST for determining header conventions.
--log-time                 Add date/time to log messages.

```

4.2.1 File Oriented Best References

The most common end-user use case for `crds.bestrefs` is to assign best references to the header keywords of dataset FITS files. This can be done as follows:

```
% python -m crds.bestrefs --update-bestrefs --sync-references=1 --files j8bt05njq_raw.fits j8bt06o6q_raw.fits
```

which will use the CRDS rules currently operational in the pipeline and download any required CRDS rules and reference files to your CRDS cache automatically. No download should occur for previously cached files or the default group readonly reference cache.

A specific historical set of CRDS rules can be used by specifying `--new-context`:

```
% python -m crds.bestrefs --new-context hst_0294.pmap --update-bestrefs --sync-references=1 --files j8bt05njq_raw.fits j8bt06o6q_raw.fits
```

4.2.2 New Context

`crds.bestrefs` always computes best references with respect to a context which can be explicitly specified with the `--new-context` parameter. If `--new-context` is not specified, the default operational context is determined by consulting the CRDS server or looking in the local cache.

4.2.3 Lookup Parameter Sources

The two primary modes for `bestrefs` involve the source of reference file matching parameters. Conceptually lookup parameters are always associated with particular datasets and used to identify the references required to process those datasets.

The options `--files`, `--datasets`, `--instruments`, and `--all-instruments` determine the source of lookup parameters:

1. To find best references for a list of files do something like this:

```
% python -m crds.bestrefs --new-context hst.pmap --file j8bt05njq_raw.fits j8bt06o6q_raw.fits j8bt09jcq_raw.fits
```

the first parameter, `hst.pmap`, is the context with respect to which best references are determined.

2. To find best references for a list of catalog dataset ids do something like this:

```
% python -m crds.bestrefs --new-context hst.pmap --datasets j8bt05njq j8bt06o6q j8bt09jcq
```

3. To do mass scale testing for all cataloged datasets for a particular instrument(s) do:

```
% python -m crds.bestrefs --new-context hst.pmap --instruments acs
```

4. To do mass scale testing for all supported instruments for all cataloged datasets do:

```
% python -m crds.bestrefs --new-context hst.pmap --all-instruments
```

or to test for differences between two contexts

```
% python -m crds.bestrefs --new-context hst_0002.pmap --old-context hst_0001.pmap --all-instruments
```

4.2.4 Comparison Modes

The `--old-context` and `--compare-source-bestrefs` parameters define the best references comparison mode. Each names the origin of a set of prior recommendations and implicitly requests a comparison to the recommendations from the newly computed bestrefs determined by `--new-context`.

Context-to-Context

`--old-context` can be used to specify a second context for which bestrefs are dynamically computed; `--old-context` implies that a bestrefs comparison will be made with `--new-context`. If `--old-context` is not specified, it defaults to `None`.

Prior Source Recommendations

`--compare-source-bestrefs` requests that the bestrefs from `--new-context` be compared to the bestrefs which are recorded with the lookup parameter data, either in the file headers of data files, or in the catalog. In both cases the prior best references are recorded static values, not dynamically computed bestrefs.

4.2.5 Output Modes

`crds.bestrefs` supports several output modes for bestrefs and comparison results to standard out.

If `--print-affected` is specified, `crds.bestrefs` will print out the name of any file for which at least one update for one reference type was recommended. This is essentially a list of files to be reprocessed with new references.:

```
% python -m crds.bestrefs --new-context hst.pmap --files j8bt05njq_raw.fits j8bt06o6q_raw.fits j8bt09jcq_raw.fits
--compare-source-bestrefs --print-affected
j8bt05njq_raw.fits
j8bt06o6q_raw.fits
j8bt09jcq_raw.fits
```

4.2.6 Update Modes

`crds.bestrefs` initially supports one mode for updating the best reference recommendations recorded in data files:

```
% python -m crds.bestrefs --new-context hst.pmap --files j8bt05njq_raw.fits j8bt06o6q_raw.fits j8bt09jcq_raw.fits
--compare-source-bestrefs --update-bestrefs
```

4.2.7 Verbosity

crds.bestrefs has `--verbose` and `--verbosity=N` parameters which can increase the amount of informational and debug output.

4.2.8 Bad Files

CRDS files can be designated as scientifically invalid on the CRDS server by the CRDS team. Knowledge of bad files is synchronized to remote caches by `crds.bestrefs` and `crds.sync`. By default, attempting to use bad rules or assign bad references will generate errors and fail. `crds.bestrefs` supports two command line switches, `--allow-bad-rules` and `--allow-bad-references` to override the default handling of bad files and enable their use with warnings. Environment variables `CRDS_ALLOW_BAD_RULES` and `CRDS_ALLOW_BAD_REFERENCES` can also be set to 1 to establish warnings rather than errors as the default.

4.3 crds.sync

The CRDS sync tool is used to download CRDS rules and references from the CRDS server:

```
usage: /Users/jmiller/jmiller_ureka/lib/python2.7/site-packages/crds/sync.py
  [-h] [--contexts [CONTEXT [CONTEXT ...]]] [--range MIN:MAX] [--all]
  [--last-n-contexts N] [--files [FILES [FILES ...]]]
  [--dataset-files [DATASET [DATASET ...]]]
  [--dataset-ids [DATASET [DATASET ...]]] [--fetch-references]
  [--purge-references] [--purge-mappings] [--dry-run] [-k] [-s] [-r]
  [--purge-rejected] [--purge-blacklisted] [--fetch-sqlite-db]
  [--organize [NEW_SUBDIR_MODE]] [--organize-delete-junk] [-v]
  [--verbosity VERBOSITY] [-R] [-I] [-V] [-J] [-H] [--stats]
  [--profile PROFILE] [--log-time] [--pdb]
```

Synchronize local mapping and reference caches for the given contexts by downloading missing files from the CRDS server and/or archive.

optional arguments:

```
-h, --help                show this help message and exit
--contexts [CONTEXT [CONTEXT ...]]
                           Specify a list of CRDS mappings to operate on: .pmap, .imap, or .rmap or date-l
--range MIN:MAX           Operate for pipeline context ids (.pmaps) between <MIN> and <MAX>.
--all                     Operate with respect to all known CRDS contexts.
--last-n-contexts N       Operate with respect to the last N contexts.
--files [FILES [FILES ...]]
                           Explicitly list files to be synced.
--dataset-files [DATASET [DATASET ...]]
                           Cache references for the specified datasets FITS files.
--dataset-ids [DATASET [DATASET ...]]
                           Cache references for the specified dataset ids.
--fetch-references         Cache all the references for the specified contexts.
--purge-references         Remove reference files not referred to by contexts from the cache.
--purge-mappings           Remove mapping files not referred to by contexts from the cache.
--dry-run                 Don't remove purged files, or repair files, just print out their names.
-k, --check-files          Check cached files against the CRDS database and report anomalies.
-s, --check-shalsum        For --check-files, also verify file shalsums.
-r, --repair-files         Repair or re-download files noted as bad by --check-files
--purge-rejected           Purge files noted as rejected by --check-files
--purge-blacklisted       Purge files (and their mapping ancestors) noted as blacklisted by --check-file
```

```
--fetch-sqlite-db      Download a sqlite3 version of the CRDS file catalog.
--organize [NEW_SUBDIR_MODE]
                        Migrate cache to specified structure, 'flat' or 'instrument'. Perform only on 1
--organize-delete-junk
                        When --organize'ing, delete obstructing files or directories CRDS discovers.
-v, --verbose          Set log verbosity to True, nominal debug level.
--verbosity VERBOSITY
                        Set log verbosity to a specific level: 0..100.
-R, --readonly-cache   Don't modify the CRDS cache. Not compatible with options which implicitly mod
-I, --ignore-cache     Download required files even if they're already in the cache.
-V, --version          Print the software version and exit.
-J, --jwst            Force observatory to JWST for determining header conventions.
-H, --hst             Force observatory to HST for determining header conventions.
--stats               Track and print timing statistics.
--profile PROFILE     Output profile stats to the specified file.
--log-time            Add date/time to log messages.
--pdb                Run under pdb.
```

- Dry-Running Cache Changes

Since CRDS cache operations can involve significant network downloads, as a general note, `crds.sync` can be run with `--readonly-cache --verbose` switches to better determine what the effects of any command should be. This can be used to gauge download sizes or list files before deleting them.

- Syncing Specific Files

Downloading an explicit list of files can be done by like this:

```
% python -m crds.sync --files hst_0001.pmap hst_acs_darkfile_0037.fits
```

this will download only those two files.

- Syncing Rules

Typically syncing CRDS files is done with respect to particular CRDS contexts:

Synced contexts can be explicitly listed:

```
% python -m crds.sync --contexts hst_0001.pmap hst_0002.pmap
```

this will recursively download all the mappings referred to by .pmaps 0001 and 0002.

Synced contexts can be specified as a numerical range:

```
% python -m crds.sync --range 1:3
```

this will also recursively download all the mappings referred to by .pmaps 0001, 002, 0003.

Synced contexts can be specified as `--all` contexts:

```
% python -m crds.sync --all
```

this will recursively download all CRDS mappings for all time.

- Syncing References By Context

Because complete reference downloads can be enormous, you must explicitly specify when you wish to fetch the references which are enumerated in particular CRDS rules:

```
% python -m crds.sync --contexts hst_0001.pmap hst_0002.pmap --fetch-references
```

will download all the references mentioned by contexts 0001 and 0002.

This can be a huge (1T+) network download and should generally only be used by institutions, not individual researchers.

NOTE: the contexts synced can be for particular instruments or types rather than the entire pipeline, e.g. `hst_cos_0002.imap` or `hst_cos_proftab_0001.rmap`

- Removing Unused Files

CRDS rules from **unspecified** contexts can be removed like this:

```
% python -m crds.sync --contexts hst_0004.pmap hst_0005.pmap --purge-mappings
```

while this would remove references which are *not* in contexts 4 or 5:

```
% python -m crds.sync --contexts hst_0004.pmap hst_0005.pmap --purge-references
```

Again, both of these commands remove cached files which are not specified or implied.

- References for Dataset Files

References required by particular dataset files can be cached like this:

```
% python -m crds.sync --contexts hst_0001.pmap hst_0002.pmap --dataset-files <dataset_file>
```

This will fetch all the references required to support the listed datasets for contexts 0001 and 0002.

This mode does not update dataset file headers. See also `crds.bestrefs` for similar functionality with header updates.

- References for Dataset Ids

References for particular dataset ids can be cached like this:

```
% python -m crds.sync --contexts hst_0001.pmap hst_0002.pmap --dataset-ids <ids...> e.g.
```

This will fetch all the references required to support the listed dataset ids for contexts 0001 and 0002.

- Checking and Repairing Large Caches

Large Institutional caches can be checked and/or repaired like this:

```
% python -m crds.sync --contexts hst_0001.pmap --fetch-references --check-sha1sum --repair-f
```

will download all the files in `hst_0001.pmap` not already present.

Both mappings and references would then be checked for correct length, sha1sum, and status.

Any files with bad length or checksum would then be deleted and re-downloaded. This is really intended for a large *existing* cache.

File checksum verification is optional because it is time consuming. Verifying the contents of the current HST shared cache requires 8-10 hours. In contrast, doing simple length, existence, and status checks takes 5-10 minutes, sufficient for a quick check but not foolproof.

- Checking Smaller Caches, Identifying Foreign Files

The simplest approach for “repairing” a small cache is to delete it and resync. One might do this after making temporary modifications to cached files to return to the archived version:

```
% rm -rf $CRDS_PATH
% python -m crds.sync -- ... # repeat whatever syncs you did to cache files of interest
```

A more complicated but also more precise approach can operate only on files already in the CRDS cache:

```
% python -m crds.sync --repair-files --check-sha1sum --files `python -m crds.list --all --ca
```

This approach works by using the `crds.list` command to dump the file names of all files in the CRDS cache and then using the `crds.sync` command to check exactly those files.

Since `crds.list` will print the name of any file in the cache, not just files from CRDS, the second approach can also be used to detect (most likely test) files which are not from CRDS.

For smaller caches `--check-sha1sum` is likely to be less of a performance/runtime issue and should be used to detect files which have changed in contents but not in length.

- Removing Blacklisted or Rejected Files

`crds.sync` can be used to remove the files from specific contexts which have been marked as “bad”.

```
% python -m crds.sync -contexts hst_0001.pmap -fetch-references -check-files -purge-  
rejected -purge-blacklisted
```

would first sync the cache downloading all the files in `hst_0001.pmap`. Both mappings and references would then be checked for correct length. Files reported as rejected or blacklisted by the server would be removed.

- Reorganizing Cache Structure

CRDS now supports two cache structures for organizing references: *flat* and *instrument*. *flat* places all references for a telescope in a single directory, e.g. `references/hst`. *instrument* segregates references into subdirectories which name instruments or legacy environment variables, e.g. `acs` or `jref`.

Newly created caches will default to the *instrument* organization. To migrate a legacy cache with a flat single directory layout to the new structure, sync with `--organize=instrument`:

```
% python -m crds.sync --organize=instrument --verbose
```

To migrate to the flat structure, use `--organize=flat`:

```
% python -m crds.sync --organize=flat --verbose
```

While reorganizing, if CRDS makes note of “junk files” in your cache which are obstructing the process of reorganizing, you can allow CRDS to delete the junk by adding `--organize-delete-junk`.

The `--organize` switches are intended to be used only on inactive file caches when calibration software is not running and actively using CRDS.

4.4 crds.certify

`crds.certify` checks a reference or mapping file against constraints on legal matching parameter values. For reference files, `crds.certify` also performs checks of the FITS format and when given a context, and will compare the given file against the file it replaces looking for new or missing table rows.

- `crds.certify --help` yields:

```
usage: /Users/jmiller/work/workspace_crds/CRDS/crds/certify.py  
[-h] [-d] [-r] [-a] [-e] [-p] [-x COMPARISON_CONTEXT]  
[-y COMPARISON_REFERENCE] [-s] [--dump-unique-errors]  
[--unique-errors-file UNIQUE_ERRORS_FILE]  
[--all-errors-file ALL_ERRORS_FILE] [-v] [--verbosity VERBOSITY] [-R]  
[-I] [-V] [-J] [-H] [--stats] [--profile PROFILE] [--log-time] [--pdb]  
[--debug-traps]  
files [files ...]
```

- Checks a CRDS reference or mapping file:

1. Verifies basic file format: .fits, .json, .yaml, .asdf, .pmap, .imap, .rmap
2. Checks references for required keywords and values, where constraints are defined.
3. Checks CRDS rules for permissible values with respect to defined reference constraints.
4. Checks CRDS rules for accidental file reversions or duplicate lines.
5. Checks CRDS rules for noteworthy version-to-version changes such as new or removed match cases.
6. Checks tables for deleted or duplicate rows relative to a comparison table.
7. Finds comparison references with respect to old CRDS contexts.

- positional arguments:

files

- optional arguments:

```
-h, --help                show this help message and exit
-d, --deep                Certify reference files referred to by mappings have valid contents.
-r, --dont-recurse-mappings Do not load and validate mappings recursively, checking only direct
-a, --dont-parse          Skip slow mapping parse based checks, including mapping duplicate entry checks
-e, --exist               Certify reference files referred to by mappings exist.
-p, --dump-provenance     Dump provenance keywords.
-x COMPARISON_CONTEXT, --comparison-context COMPARISON_CONTEXT Pipeline context defining comparison
-y COMPARISON_REFERENCE, --comparison-reference COMPARISON_REFERENCE Comparison reference for table
-s, --sync-files          Fetch any missing files needed for the requested difference from the CRDS
-v, --verbose             Set log verbosity to True, nominal debug level.
--verbosity VERBOSITY    Set log verbosity to a specific level: 0..100.
-R, --readonly-cache     Don't modify the CRDS cache. Not compatible with options which implicitly
```

- crds.certify is normally invoked as, e.g.:

```
% python -m crds.certify --comparison-context=hst_0027.pmap some_reference.fits
% python -m crds.certify hst.pmap
```

- To run crds.certify on a reference(s) to verify basic file format and parameter constraints:

```
% python -m crds.certify --comparison-context=hst_0027.pmap some_reference.fits...
```

If some_reference.fits is a table, a comparison table will be found in the comparison context, if appropriate.

- For recursively checking CRDS rules do this:

```
% python -m crds.certify hst_0311.pmap --comparison-context=hst_0312.pmap
```

If a comparison context is defined, checked mappings will be compared against their peers (if they exist) in the comparison context. Many classes of mapping differences will result in warnings.

- For reference table checks, a comparison reference can also be specified directly rather than inferred from context:

```
% python -m crds.certify some_reference.fits --comparison-reference=old_reference_version.fits
```

- For more information on the checks being performed, use `-verbose` or `-verbosity=N` where $N > 50$.
- Invoking crds.certify on a context mapping recursively certifies all sub-mappings.

4.5 crds.diff

crds.diff compares two reference or mapping files and reports differences. For references crds.diff is currently a thin wrapper around fitsdiff but may expand.

For CRDS mappings crds.diff performs a recursive logical difference which shows the full match path to each bottom level change. crds.diff --help yields:

Difference CRDS mapping or reference files.

positional arguments:

old_file	Prior file of difference.
new_file	New file of difference.

optional arguments:

-h, --help	show this help message and exit
-P, --primitive-diffs	Fitsdiff replaced reference files when diffing mappings.
-T, --mapping-text-diffs	In addition to CRDS mapping logical differences, run UNIX context diff for mappings.
-K, --check-diffs	Issue warnings about new rules, deletions, or reversions.
-N, --print-new-files	Rather than printing diffs for mappings, print the names of new or replacement files.
-A, --print-all-new-files	Print the names of every new or replacement file in diffs between old and new mappings.
-i, --include-header-diffs	Include mapping header differences in logical diffs: shalsum, derived_from, etc.
-B, --hide-boring-diffs	Include mapping header differences in logical diffs: shalsum, derived_from, etc.
--print-affected-instruments	Print out the names of instruments which appear in diffs, rather than diffing them.
--print-affected-types	Print out the names of instruments and types which appear in diffs, rather than diffing them.
--print-affected-modes	Print out the names of instruments, types, and matching parameters, rather than diffing them.
-v, --verbose	Set log verbosity to True, nominal debug level.
--verbosity VERBOSITY	Set log verbosity to a specific level: 0..100.
-R, --readonly-cache	Don't modify the CRDS cache. Not compatible with options which implicitly modify the cache.
-V, --version	Print the software version and exit.
-J, --jwst	Force observatory to JWST for determining header conventions.
-H, --hst	Force observatory to HST for determining header conventions.

Reference files are nominally differenced using FITS-diff or diff.

Mapping files are differenced using CRDS machinery to recursively compare two mappings and their sub-mappings.

Differencing two mappings will find all the logical differences between the two contexts and any nested mappings.

By specifying --mapping-text-diffs, UNIX diff will be run on mapping files in addition to CRDS logical diffs.

By specifying --primitive-diffs, FITS diff will be run on all references which are replaced in the logical differences between two mappings.

For example:


```
% python -m crds.diff hst_0001.pmap hst_0005.pmap --mapping-text-diffs --primitive-diffs
```

Will recursively produce logical, textual, and FITS diffs for all changes between the two contexts.

NOTE: mapping logical differences (the default) do not compare CRDS mapping headers, use `--include-header-diffs` to get those as well.

For standard CRDS filenames, `crds.diff` can guess the observatory. For non-standard names, the observatory needs to be specified. `crds.diff` can be invoked like:

```
% python -m crds.diff jwst_nircam_dark_0010.fits jwst_nircam_dark_0011.fits
```

```
% python -m crds.diff jwst_0001.pmap jwst_0002.pmap
(('hst.pmap', 'hst_0004.pmap'), ('hst_acs.imap', 'hst_acs_0004.imap'), ('hst_acs_darkfile.rmap', 'hst_acs_darkfile_0004.rmap'))
```

4.6 crds.rowdiff

Modules that are based on `FITSDiff`, such as `crds.diff`, compare tabular data on a column-by-column basis. `Rowdiff` compares tabular data on a row-by-row basis, producing UNIX diff-like output instead. Non-tabular extensions are ignored.

usage: `rowdiff.py [-J] [-H]`

`[-ignore-fields IGNORE_FIELDS] [-fields FIELDS] [-mode-fields MODE_FIELDS] old_file new_file`

Perform FITS table difference by rows

positional arguments:

`old_file` First FITS table to compare `new_file` Second FITS table to compare

optional arguments:

--ignore-fields IGNORE_FIELDS

List of fields to ignore

--fields FIELDS

List of fields to compare

--mode-fields MODE_FIELDS

List of fields to do a mode compare

-J, --jwst

Force observatory to JWST for determining header conventions.

-H, --hst

Force observatory to HST for determining header conventions.

The FITS data to be compared are required to be similar: they must have the same number of extensions and the types of extensions must match.

The parameters `--fields` and `--ignore-fields` define which columns are compared between each table extension. These are mutually exclusive parameters and an error will generate if both are specified.

First a summary of the changes between the table extension is given. Then, row-by-row difference is given, using unified diff syntax.

The parameter `--mode-fields` initiates a different algorithm. Here, it is presumed the tabular data contains columns that can essentially be treated as keys upon which rows are selected. The fields specified are those key columns.

All possible combinations of values are determined by examining both extensions. Then, each table is compared against both this list and between each other, looking for multiply specified combinations, missing combinations, and, for the common combinations between the tables, whether the rest of the rows are equivalent or not.

Examples:

```
% python -m crds.rowdiff s9m1329lu_off.fits s9518396u_off.fits
% python -m rowdiff s9m1329lu_off.fits s9518396u_off.fits --mode-fields=detchip,obsdate
```

4.7 crds.uses

crds.uses searches the files in the local cache for mappings which refer to the specified files. Since the **local cache** is used only mappings present in the local cache will be included in the results given. crds.uses is invoked as:

```
% python -m crds.uses <observatory=hst|jwst> <mapping or reference>...
```

e.g.:

Prints out the mappings which refer to the specified mappings or references.

Prints out the datasets which historically used a particular reference as defined by DADSOPS.

IMPORTANT:

1. You must specify references on which to operate with `--files`.
2. You must set `CRDS_PATH` and `CRDS_SERVER_URL` to give crds.uses access to CRDS mappings and datasets.

optional arguments:

<code>-h, --help</code>	show this help message and exit
<code>--files FILES [FILES ...]</code>	References for which to dump using mappings or datasets.
<code>-d, --print-datasets</code>	Print the ids of datasets last historically using a reference.
<code>-i, --include-used</code>	Include the used file in the output as the first column.
<code>-v, --verbose</code>	Set log verbosity to True, nominal debug level.
<code>--verbosity VERBOSITY</code>	Set log verbosity to a specific level: 0..100.
<code>-R, --readonly-cache</code>	Don't modify the CRDS cache. Not compatible with options which implicitly modify the cache.
<code>-V, --version</code>	Print the software version and exit.
<code>-J, --jwst</code>	Force observatory to JWST for determining header conventions.
<code>-H, --hst</code>	Force observatory to HST for determining header conventions.

crds.uses can be invoked like this:

```
% python -m crds.uses --files n3o1022ij_drk.fits --hst
hst.pmap
hst_0001.pmap
hst_0002.pmap
hst_0003.pmap
...
hst_0041.pmap
hst_acs.imap
hst_acs_0001.imap
hst_acs_0002.imap
hst_acs_0003.imap
...
hst_acs_0008.imap
hst_acs_darkfile.rmap
hst_acs_darkfile_0001.rmap
```

```

hst_acs_darkfile_0002.rmap
hst_acs_darkfile_0003.rmap
...
hst_acs_darkfile_0005.rmap

% python -m crds.uses --files n3o1022ij_drk.fits --print-datasets --hst
J8BA0HRPQ
J8BA0IRTQ
J8BA0JRWQ
J8BA0KT4Q
J8BA0LIJQ

% python -m crds.uses --files @dropped --hst --print-datasets --include-used
vb41934lj_bia.fits JA7P21A2Q
vb41934lj_bia.fits JA7P21A4Q
vb41934lj_bia.fits JA7P21A6Q

```

4.8 crds.matches

`crds.matches` reports the match patterns which are associated with the given reference files:

```

usage: matches.py
       [-h] [--contexts [CONTEXT [CONTEXT ...]]]
       [--files FILES [FILES ...]] [-b] [-o] [-t]

```

Prints out the selection criteria by which the specified references are matched with respect to a particular context.

optional arguments:

```

-h, --help                show this help message and exit
--contexts [CONTEXT [CONTEXT ...]]
                           Specify a list of CRDS mappings to operate on: .pmap, .imap, or .rmap or dataset
--range MIN:MAX            Operate for pipeline context ids (.pmaps) between <MIN> and <MAX>.
--all                     Operate with respect to all known CRDS contexts.
--last N                   Operate with respect to the last N contexts.
-i, --ignore-cache        Download required files even if they're already in the cache.
--files FILES [FILES ...]
                           References for which to dump selection criteria.
-b, --brief-paths         Don't the instrument and filekind.
-o, --omit-parameter-names
                           Hide the parameter names of the selection criteria, just show the values.
-t, --tuple-format        Print the match info as Python tuples.
-d DATASETS [DATASETS ...], --datasets DATASETS [DATASETS ...]
                           Dataset ids for which to dump matching parameters from DADSOPS or equivalent
-c, --condition-values    When dumping dataset parameters, first apply CRDS value conditioning / normalization
-m, --minimize-header     When dumping dataset parameters, limit them to matching parameters, not histogram
-v, --verbose             Set log verbosity to True, nominal debug level.
--verbosity VERBOSITY     Set log verbosity to a specific level: 0..100.
-R, --readonly-cache      Don't modify the CRDS cache. Not compatible with options which implicitly modify
-V, --version             Print the software version and exit.
-J, --jwst               Force observatory to JWST for determining header conventions.
-H, --hst                Force observatory to HST for determining header conventions.

```

crds.matches can dump reference file match cases with respect to particular contexts:

```
% python -m crds.matches --contexts hst_0001.pmap --files lc41311jj_pfl.fits
lc41311jj_pfl.fits : ACS PFLTFILE DETECTOR='WFC' CCDAMP='A|ABCD|AC|AD|B|BC|BD|C|D' FILTER1='F625W' F

% python -m crds.matches --contexts hst.pmap --files lc41311jj_pfl.fits --omit-parameter-names --brief
lc41311jj_pfl.fits : 'WFC' 'A|ABCD|AC|AD|B|BC|BD|C|D' 'F625W' 'POL0V' '1997-01-01' '00:00:00'

% python -m crds.matches --contexts hst.pmap --files lc41311jj_pfl.fits --tuple-format
lc41311jj_pfl.fits : (('OBSERVATORY', 'HST'), ('INSTRUMENT', 'ACS'), ('FILEKIND', 'PFLTFILE'), ('DET
```

crds.matches can dump database matching parameters for specified datasets with respect to specified contexts:

```
% python -m crds.matches --datasets JBANJOF3Q --minimize-headers --contexts hst_0048.pmap hst_0044.p
JBANJOF3Q : hst_0044.pmap : APERTURE='WFC1-2K' ATODCORR='NONE' BIASCORR='NONE' CCDAMP='B' CCDCHIP='1
JBANJOF3Q : hst_0048.pmap : APERTURE='WFC1-2K' ATODCORR='NONE' BIASCORR='NONE' CCDAMP='B' CCDCHIP='1
```

crds.matches can be invoked in various ways with different output formatting:

```
% python -m crds.matches --contexts hst_0001.pmap --files lc41311jj_pfl.fits
lc41311jj_pfl.fits : ACS PFLTFILE DETECTOR='WFC' CCDAMP='A|ABCD|AC|AD|B|BC|BD|C|D' FILTER1='F625W' F

% python -m crds.matches --contexts hst.pmap --files lc41311jj_pfl.fits --omit-parameter-names --brief
lc41311jj_pfl.fits : 'WFC' 'A|ABCD|AC|AD|B|BC|BD|C|D' 'F625W' 'POL0V' '1997-01-01' '00:00:00'

% python -m crds.matches --contexts hst.pmap --files lc41311jj_pfl.fits --tuple-format
lc41311jj_pfl.fits : (('OBSERVATORY', 'HST'), ('INSTRUMENT', 'ACS'), ('FILEKIND', 'PFLTFILE'), ('DET
```

4.9 pipeline_bestrefs

The pipeline_bestrefs script is a shim around crds.bestrefs which simplifies the command line interface, tuning it to the more limited case of updating FITS dataset headers with best references:

```
usage: pipeline_bestref [-d] [-v] [-h] [--print-affected] <crds_context> <dataset_file(s)>...
```

```
-d                dry run, do not update file headers
-v                verbose, output additional diagnostic messages
-h                help, print this help
--print-affected  print files with updated bestrefs
```

Updates dataset FITS files with best references recommended by <crds_context>.

<crds_context> is a CRDS context file, explicitly named e.g. hst_0004.pmap
<crds_context> can be specified abstractly, e.g. hst-edit or hst-operational
<crds_context> can be specified by date, e.g. hst-2013-01-29T12:00:00

<dataset_file(s)> are raw dataset files for which best references are computed and updated.

LIBRARY 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.

5.1 `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,
                  observatory="jwst"):
    """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.

observatory : The name of the observatory this query applies to, needed
              to support both 'hst' and 'jwst' from a single server.

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',
    }
"""
```

5.2 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'
    """
```

CORE LIBRARY FUNCTIONS

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.

6.1 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.

6.2 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

6.3 Basic Operations on Mappings

6.3.1 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 its 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.

6.3.2 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 its 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_cfltfilermap',
 '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_crrehtab.rmap',
'hst_acs_pfltfile.rmap',
'hst_acs_biasfile.rmap',
'hst_acs_mdrihtab.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']
```

6.3.3 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 to 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',
'OBSERVE': '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',
'crrehtab': 'n4e12510j_crr.fits',
'darkfile': 'n3o1059hj_drk.fits',
'dgeofile': '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.

6.4 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.

6.4.1 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)
```

Alternately you can set an environment variable to ignore the mapping checksum while you iterate on new versions of the mapping:

```
% setenv CRDS_IGNORE_MAPPING_CHECKSUM 1
```

6.4.2 Adding Checksums

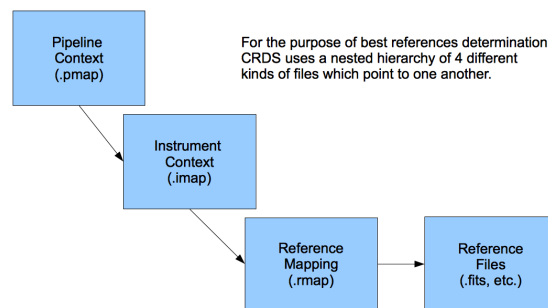
Once you've finished your masterpiece ReferenceMapping, it can be sealed with a checksum like this:

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


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.



7.1 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

7.2 Basic Structure

All mappings have the same basic structure consisting of a “header” section followed by a “selector” section.

7.2.1 header

The header provides meta data describing the mapping. A critical field in the mapping header is the “parkey” field which names the dataset parameters (nominally FITS keywords or JWST data model names) which are used by the selector to do a best references lookup.

7.2.2 selector

The selector provides matching rules used to look up the results of the mapping. The selector is a nested tree structure consisting of top-level selectors and sub-selectors.

7.3 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',
    'shasum' : '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.

7.4 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',
    'flxstab' : 'hst_cos_flxstab.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.

7.5 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.

7.6 Active Header Fields

Many rmap header fields are passive metadata. A number of optional rmap header fields, however, actively affect best reference lookups and results:

```
header = {
    ...,

    'parkey' : (('DETECTOR',), ('DATE-OBS', 'TIME-OBS')),

    'extra_keys' : ('XCORNER', 'YCORNER', 'CCDCHIP'),

    'reffile_switch' : 'BIASCORR',

    'reffile_required' : 'YES',

    'rmap_relevance' : '((DETECTOR != "SBC") and (BIASCORR != "OMIT"))',
    'rmap_omit' : '((DETECTOR != "SBC") and (BIASCORR != "OMIT"))',

    'parkey_relevance' : {
        'binaxis1' : '(DETECTOR == "UVIS")',
        'binaxis2' : '(DETECTOR == "UVIS")',
        'ccdgain' : '(DETECTOR == "IR")',
        'samp_seq' : '(DETECTOR == "IR")',
        'subtype' : '(DETECTOR == "IR")',
    },

    'hooks' : {
        'fallback_header' : 'fallback_header_acs_biasfile_v2',
        'precondition_header' : 'precondition_header_acs_biasfile_v2',
    },

    ...,
}
```

7.6.1 Required Parameters

Required matching parameters for computing best references are defined by the union of 3 header fields: *parkey*, *extra_keys*, and *reffile_switch*. There is no requirement to use all 3 forms, the latter two forms were added to model and emulate aspects of HST's CDBS system, the precursor to CRDS.

parkey

The primary location for defining best references matching parameters is the *parkey* field.

The simplest form of *parkey* is a tuple of parameter names used in a lookup by a non-nested selector, as is seen in pipeline and instrument mappings above.

In reference mappings, the header *parkey* field is a tuple of tuples. Each stage of the nested selector consumes the next tuple of header keys. The same parameter set and matching structure is shared by all sections of a single rmap. For mode-specific parameters, two approaches are available: use a separate .rmap for each parameter combination, or fill in unused parameters for a particular mode with the value 'N/A'.

For the HST COS DEADTAB 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.

There is no default for parkey.

extra_keys

extra_keys specifies a tuple of parameter names which will not be used in the matches directly, but may be used by rmap header expressions and hook functions to influence matching. Listing parameters in *extra_keys* ensures that the CRDS infrastructure will request the parameters from the server or dataset files and make them available during best references computations and logical expression evaluation. All parameters used in logical expressions must be explicitly defined and listed. Undefined parameters are evaluated with the value 'UNDEFINED'.

If omitted, *extra_keys* defaults to (), no extra keys.

reffile_switch

Nominally names a dataset keyword generally of the form <type>CORR with keyword values 'PERFORM' and 'OMIT'.

If *reffile_switch* is not 'NONE', it specifies an extra keyword value is to fetch from the dataset.

If *reffile_switch* is omitted or 'NONE', no keyword value is fetched from the dataset.

The runtime checking *reffile_switch* is used for must be explicitly implemented as part of an *rmap_relevance* or *rmap_omit* expression as seen in the example header; *reffile_switch* only specifies an extra parameter to fetch for use in logical expressions and matching. It is logically equivalent to adding the parameter to *extra_keys*.

7.6.2 Logical Header Expressions

A number of the subsequently described features employ logical expressions which are evaluated at match-time based on the values in the dataset header. There are several things to point out:

- Logical expressions are evaluated in the context of the required parameters discussed above.
- Dataset matching parameters appear in logical expressions in upper case, without quotes, like global variables.
- The entire expression is enclosed in parentheses to tell CRDS to leave case as-is.
- Logical expressions are limited to a restricted subset of Python expressions, not arbitrary Python. In particular arbitrary Python function calls are not permitted.

7.6.3 reffile_required

Defines what should happen if an rmap lookup cannot find a match for a particular reference type.

reffile_required has legal values 'YES', 'NO', and 'NONE'.

If *reffile_required* is 'YES', failing to find a match results in an exception and/or ERROR.

If *reffile_required* is 'NONE', CRDS did not define *reffile_required* for this type, so it is assumed to be required.

If *reffile_required* is 'NO', failing to find a match results in assigning the value 'N/A' rather than failing.

7.6.4 rmap_relevance

rmap_relevance is a logical expression which is evaluated in the context of dataset header variables.

If *rmap_relevance* evaluates to True, then a full match is performed and the resulting bestref is returned.

If *rmap_relevance* evaluates to False, then the match is short circuited and 'N/A' is assigned.

7.6.5 parkey_relevance

parkey_relevance defines a mapping from dataset matching parameters to logical expressions.

parkey_relevance is evaluated in the context of the entire set of matching parameters and mutates the specified parameter to 'N/A' if the expression evaluates to False, i.e. the parameter is not relevant in the context of the other parameter values.

When a parameter value of 'N/A' is used for matching, the parameter is effectively ignored.

7.6.6 hooks

The *hooks* header section defines functions which are used for special case processing for complex reference assignments. The existing hooks were devised to emulate similar special case handling performed by CRDS's predecessor system CDBS.

The original <100 series of HST rules had implicit hooks. CRDS rules >200 have hooks which are explicitly named in the 'hooks' section of the header which indicates that customized matching is being performed. Running `crds.bestrefs` with `-verbosity=60` will issue log messages describing hook operations.

new hook functions can only be added with a new release of CRDS code. hook functions have versioned names and should never be modified after use in operations since that would change the meaning of historical `.rmaps`. Instead, a new hook function should be added and the `.rmap` header modified to assign it.

hook functions can be 'unplugged' in an operational `.rmap` by setting the value of the hook to 'none'. Removing the 'hooks' section of the `.rmap` header, or removing individual hook names, currently results in reversion to <100 series `.rmap` behavior and the original implicit hook functions.

precondition_header

The *precondition_header* hook is used to mutate incoming dataset matching parameters. *precondition_header* is sometimes justified as reductive, written in terms of *extra_parkeys* which do not appear in the matching tuples, and used to mutate a broad range of matching parameter values onto a narrower set of parameter values known to be handled in the `.rmap`. In essence, when a *precondition_header* hook is used, the dataset matching parameters become a function of themselves.

fallback_header

The *fallback_header* hook is used to mutate incoming dataset matching parameters similar to *precondition_header*. The *fallback_header* hook is called when the first matching attempt for dataset parameters fails. *fallback_header* computes a set of matching parameters used for a second matching attempt which will return normally if successful.

7.7 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.

7.7.1 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 {}, Relational operators <, >, <=, >=, 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[1234]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

Substitution 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 accommodated by merging or eventually addressed by human beings who will simplify the rules by hand.

7.7.2 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 (\leq) 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'

7.7.3 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.

7.7.4 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'.

7.7.5 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'.

7.7.6 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')
```


USING THE CRDS WEB SITE

CRDS has websites at hst-crds.stsci.edu and jwst-crds.stsci.edu which support the submission, use, and distribution of CRDS reference and mappings files. Functions on the CRDS website are either public functions which do not require authentication or private functions which require a CRDS login account.

HST Calibration Reference Data System (CRDS) (dev)

Operational References
(under context [hst_0271.pmap](#))

- acs
- cos
- nicos
- stis
- wfc3
- wfc2

Context History ([more](#))

Start Date	Context	Status	Description
2014-08-18	hst_0271.pmap	operational	New ACS BIAS FILES wfc1-1k_gain2_140803_bia.fits wfc1-2k_gain2_140516_bia.fits
2014-08-11	hst_0270.pmap	archived	Delivery of new WFC3/UVIS post-flashed darks.
2014-08-05	hst_0269.pmap	archived	New STIS darks and biases delivered for the new anneal month.
2014-07-15	hst_0268.pmap	archived	Delivery of new WFC3 UVIS post-flashed dark files.

Services

- [Dataset Best References](#)
- [Explore Best References](#)
- [Browse Database](#)
- [Recent Activity](#)

Functions annotated with the word (alpha) are partially completed components of a future build which may prove useful now.

8.1 Operational References

The *Operational References* table displays the references which are currently in use by the pipeline associated with this web site. The operational context is displayed as a link '(under context <link>)' immediately below Operational References. Clicking the link opens a details browser for that CRDS .pmap reference assignment rules file. The operational context is the latest context in the Context History, the one in active use for pipeline processing by default.

Each instrument accordion opens into reference type accordions for that instrument.

Each type accordion opens into a table of reference files and the dataset parameters they apply to. Each reference file link opens into a details browser for that reference file.

8.2 Context History (more)

The *Context History* displays the last 4 CRDS contexts which were in operational use by the pipeline associated with a CRDS server. Clicking on the (*more*) link will bring up the entire context history as a separate page as shown below:

Context History				
Start Date	Context	Status	Description	diff
2014-08-18	hst_0271.pmap	operational	New ACS BIAS FILES wfc1-1k_gain2_140803_bia.fits wfc1-2k_gain2_140516_bia.fits	<input type="checkbox"/>
2014-08-11	hst_0270.pmap	archived	Delivery of new WFC3/UVIS post-flashed darks.	<input type="checkbox"/>
2014-08-05	hst_0269.pmap	archived	New STIS darks and biases delivered for the new anneal month.	<input type="checkbox"/>
2014-07-15	hst_0268.pmap	archived	Delivery of new WFC3 UVIS post-flashed dark files.	<input type="checkbox"/>
2014-07-15	hst_0267.pmap	archived	New ACS full frame darks, cte corrected darks, and biases.	<input type="checkbox"/>
2014-07-08	hst_0266.pmap	archived	The COS Team delivered a new HVTAB.	<input type="checkbox"/>
2014-07-02	hst_0265.pmap	archived	Delivery of a new COS NUV High Voltage Reference Table.	<input type="checkbox"/>
2014-07-01	hst_0264.pmap	archived	Delivery of a new COS WCPTAB and new WFC3 UVIS darks and IR BPIXTAB reference files.	<input type="checkbox"/>
2014-06-19	hst_0262.pmap	archived	The ACS Team delivered a new set of dark, cte-corrected dark, and bias reference files.	<input type="checkbox"/>
2014-06-18	hst_0261.pmap	archived	Delivery of new WFC3 dark reference files	<input type="checkbox"/>
2014-06-17	hst_0260.pmap	archived	Delivery of new STIS bias and dark files.	<input type="checkbox"/>

Clicking on a *context* link (the .pmap name) opens a page containing the Historical References for some point in the past, similar to the Operational References display:

HST Calibration Reference Data System (CRDS) (dev)

Historical References

(under context [hst_0270.pmap](#))

- acs
- cos
- nicmos
- stis
- wfc3
- wfpc2

hst_wfpc2_0250.imap

atodfile ---- Analog To Digital Converter Lookup Table

hst_wfpc2_atodfile_0250.rmap

Show entries Search:

MODE	ATODGAIN	USEAFTER	REFERENCE	diff
AREA	7	1993-12-01 00:00:00	e1b09594u.r1h	<input type="checkbox"/>
AREA	15	1993-12-01 00:00:00	e1b09593u.r1h	<input type="checkbox"/>
FULL	7	1993-12-01 00:00:00	dbu14059u.r1h	<input type="checkbox"/>

References are displayed in accordion panels for each instrument. Opening the panel for an instrument displays the reference types of that instrument. Opening the panel for a type displays particular reference files and matching parameters for that type. Clicking on a particular reference file brings up the CRDS browser page with the known details for that reference.

8.2.1 Differencing contexts

Click the *diff* checkbox for any two contexts in the history and then click the diff button at the top of the diff column. This will display a difference page with an accorion panel for each file which differed between the two contexts:

HST Calibration Reference Data System (CRDS) (dev)

Difference hst.pmap against hst_0004.pmap

- hst.pmap --> hst_0004.pmap
- hst_cos.imap --> hst_cos_0001.imap
- hst_cos_flatfile.rmap --> hst_cos_flatfile_0002.rmap
- hst_stis.imap --> hst_stis_0001.imap
- hst_stis_biasfile.rmap --> hst_stis_biasfile_0001.rmap

Logical Diff ('hst_stis_biasfile.rmap', 'hst_stis_biasfile_0001.rmap')

Show entries Search:

CCDAMP	CCDGAIN	CCDOFFST	BINAXIS1	BINAXIS2	DATE-OBS	TIME-OBS	DIFF
D	1.0	3.0	1.0	1.0	2013-04-16	22:00:01	added ter 'x8c1626'
D	1.0	3.0	1.0	1.0	2013-04-25	00:08:29	added ter 'x8c1626'
D	1.0	3.0	1.0	1.0	2013-05-02	00:00:06	added ter 'x8c1626'
D	1.0	3.0	1.0	1.0	2013-05-09	06:52:08	added ter 'x8c1626'
D	4.0	3.0	1.0	1.0	2013-04-16	22:00:01	added ter 'x8c1626'
D	4.0	3.0	1.0	1.0	2013-05-02	02:11:52	added ter 'x8c1626'

Showing 1 to 6 of 6 entries ◀ Previous Next ▶

Text Diff ('hst_stis_biasfile.rmap', 'hst_stis_biasfile_0001.rmap')

Each file accordion opens into two accordions which display different views of the differences, logical or textual. The logical differences display a table of matching parameters and files which were added, deleted, or replaced. The textual differences show raw UNIX diffs of the two rules files.

8.3 Open Services

The following functions are available for anyone with access to the CRDS web server and basically serve to distribute information about CRDS files and recommendations. Initially, the CRDS sites are only visible within the Institute.

8.3.1 Dataset Best References

The *Dataset Best References* page supports determining the best references for a single dataset with respect to one CRDS context. Best references are based upon a CRDS context and the parameters of the dataset as determined by the dataset file itself or a database catalog entry.

HST Calibration Reference Data System (CRDS) (dev)

Dataset Best References

Context:

hst_0271.pmap 2014-08-18 12:17 (Operational Context)

Dataset:

☒ Upload FITS Header No file selected.

☐ Dataset ID e.g. I9ZF01010

Dataset Best References determines the best reference files for an uploaded or cataloged dataset header.

Context defines the version of rules that should be used to determine best references.

Upload FITS Header loads reference matching parameters from a FITS file on your local system; the loaded matching parameters may be edited in the browser.

Dataset ID specifies the id of a cataloged parameter set for which to determine best references. If the dataset id names an association, the results returned are for the first member of the association.

Context

The context defines the set of CRDS rules used to select best references. *Edit* is the default context from which most newly created contexts are derived. *Operational* is the context currently in use by the pipeline. *Recent* shows the most recently created contexts. *User Specified* enables the submitter to type in the name of any other known context.

Dataset

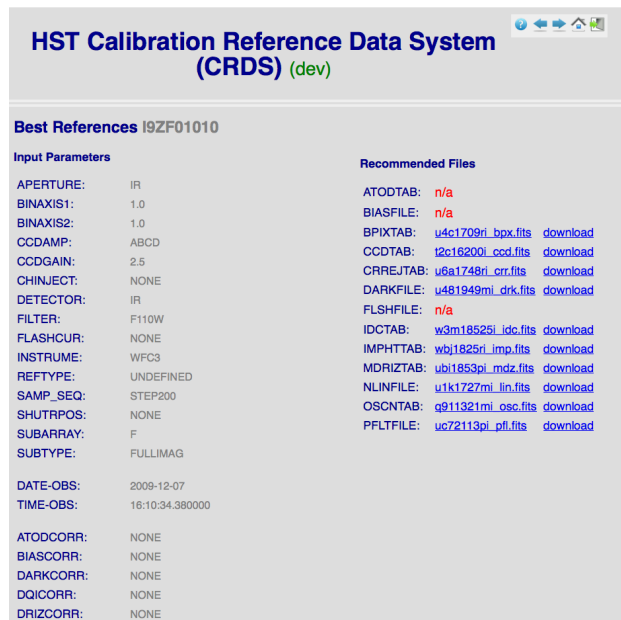
Upload FITS header

Browser-side code can extract the FITS header of a dataset and upload it to the server where best references are computed based on dataset parameters. This function is implemented in Javascript and reliant on HTML5; it supports only parameters present in the FITS primary header. It avoids uploading most of the dataset. It is known to work in Firefox and Chrome but not IE or Safari-5.

Archived Dataset

Datasets can be specified by ID and their best reference input parameters will be retrieved from the catalog.

Dataset Best References Results



The screenshot shows the HST Calibration Reference Data System (CRDS) web interface. The title is "HST Calibration Reference Data System (CRDS) (dev)". Below the title, the page is titled "Best References I9ZF01010". The page is divided into two main sections: "Input Parameters" and "Recommended Files".

Input Parameters	Recommended Files
APERTURE: IR	ATODTAB: n/a
BINAXIS1: 1.0	BIASFILE: n/a
BINAXIS2: 1.0	BPIXTAB: u4c1709ri_bpx.fits download
CCDAMP: ABCD	CCDTAB: t2c16200i_ccd.fits download
CCDGAIN: 2.5	CRREJTAB: u6a1748ri_corr.fits download
CHINJECT: NONE	DARKFILE: u481949mi_drk.fits download
DETECTOR: IR	FLSHFILE: n/a
FILTER: F110W	IDCTAB: w3m18525i_idc.fits download
FLASHCUR: NONE	IMPHTTAB: wbj1825ri_imp.fits download
INSTRUME: WFC3	MDRIZTAB: ubi1853pi_mdz.fits download
REFTYPE: UNDEFINED	NLINFILE: u1k1727mi_lin.fits download
SAMP_SEQ: STEP200	OSCNTAB: q911321mi_osc.fits download
SHUTRPOS: NONE	PFLTFILE: uc72113pi_pfl.fits download
SUBARRAY: F	
SUBTYPE: FULLIMAG	
DATE-OBS: 2009-12-07	
TIME-OBS: 16:10:34.380000	
ATODCORR: NONE	
BIASCORR: NONE	
DARKCORR: NONE	
DQICORR: NONE	
DRIZCORR: NONE	

The results page for dataset best references displays the input parameters which were extracted from the dataset header on the right side of the page.

Best reference recommendations are displayed on the left side of the page.

8.3.2 Explore Best References

Explore Best References supports entering best references parameters directly rather than extracting them from a dataset or catalog. Explore best references is essentially a sand box which lets someone evaluate what CRDS will do given particular parameter values. The explorer currently lists all parameters which might be relevant to any mode of an instrument and has no knowledge of default values.

The first phase of exploration is to choose a pipeline context and instrument which will be used to define parameter choices:

The second phase is to enter the parameters of a dataset which are relevant to best references selection.

The entered parameters are evaluated with respect to the given pipeline context and best references are determined. The results are similar or identical to the *Dataset Best References* results.

8.3.3 Browse Database

The *Browse Database* feature enables examining the metadata and computable properties of CRDS reference and mapping files.

The first phase is to enter a number of filters to narrow the number or variety of files which are displayed. Leaving any filter at the default value of * renders that constraint irrelevant and all possible files are displayed with respect to that constraint. The result of the first phase is a table of files which matched the filters showing their basic properties.

The second phase is initiated by clicking on the filename link of any file displayed in the table from the first phase. Clicking on a filename link switches to a detailed view of that file only:

The file details page has a number of accordion panes which open when you click on them. All file types have these generic panes:

- Database - lists a table of CRDS metadata for the file.

HST Calibration Reference Data System (CRDS) (dev)

Browse Database

Filters

Instrument:

Reference Type:

Filename:

Status:

Extension:

Browse Database Searches the CRDS database for files matching the criteria you specify. Filters restrict the search to files matching that value. To ignore a field during search, set it to "".

HST Calibration Reference Data System (CRDS) (dev)

Browse Database

Filters

Database Entries

Show entries

Search:

delivery date	activation date	useafter date	name	aperture	status	description	instrument	reference type
2013-07-02 15:44	2013-07-02 15:44	2050-01-01 00:00	hst_acs.imap	none	archived	Initial mass file import	acs	
2013-07-02 15:44	2013-07-02 15:44	2050-01-01 00:00	hst_acs_atodtab.rmap	none	archived	Initial mass file import	acs	atodtab
2013-07-02 15:44	2013-07-02 15:44	2050-01-01 00:00	hst_acs_biasfile.rmap	none	archived	Initial mass file import	acs	biasfile
2013-07-02 15:44	2013-07-02 15:44	2050-01-01 00:00	hst_acs_bptab.rmap	none	archived	Initial mass file import	acs	bptab
2013-07-02 15:44	2013-07-02 15:44	2050-01-01 00:00	hst_acs_codtab.rmap	none	archived	Initial mass file import	acs	codtab
2013-07-02 15:44	2013-07-02 15:44	2050-01-01 00:00	hst_acs_crtfile.rmap	none	archived	Initial mass file import	acs	crtfile
2013-07-02 15:44	2013-07-02 15:44	2050-01-01 00:00	hst_acs_corrfile.rmap	none	archived	Initial mass file import	acs	corrfile
2013-07-02 15:44	2013-07-02 15:44	2050-01-01 00:00	hst_acs_d2mfile.rmap	none	archived	Initial mass file import	acs	d2mfile
2013-07-02 15:44	2013-07-02 15:44	2050-01-01 00:00	hst_acs_darkfile.rmap	none	archived	Initial mass file import	acs	darkfile
2013-07-02 15:44	2013-07-02 15:44	2050-01-01 00:00	hst_acs_dgeofile.rmap	none	archived	Initial mass file import	acs	dgeofile
2013-07-02 15:44	2013-07-02 15:44	2050-01-01 00:00	hst_acs_drkfile.rmap	none	archived	Initial mass file import	acs	drkfile
2013-07-02 15:44	2013-07-02 15:44	2050-01-01 00:00	hst_acs_fishfile.rmap	none	archived	Initial mass file import	acs	fishfile
2013-07-02 15:44	2013-07-02 15:44	2050-01-01 00:00	hst_acs_idctab.rmap	none	archived	Initial mass file import	acs	idctab
2013-07-02 15:44	2013-07-02 15:44	2050-01-01 00:00	hst_acs_imphtab.rmap	none	archived	Initial mass file import	acs	imphtab

HST Calibration Reference Data System (CRDS) (dev)

Browse [hst_acs_darkfile.rmap](#) [download](#)

- Database
- Contents
- Past Actions
- Used By Files

- Contents - shows the text of a mapping or internal details about a reference file.
- Past Actions - lists website actions which affected this file.
- Used By Files - list known CRDS files which reference this file.

Reference files have these additional panes:

- Certify Results - shows the results of `crds.certify` run on this reference now.
- Lookup Patterns - lists the parameters sets which lead to this reference.

8.3.4 Recent Activity

The *Recent Activity* view shows a table of the actions on CRDS files which are tracked. Only actions which change the states of files in some way are tracked:

The first page lists a number of constraints which can be used to choose activities of interest. To ignore any constraint, leave it set at the default value of *. The result of the activity search is a table of matching actions:

Date	Filename	Action	Why	Details
2014-05-11 08:17	hst_acs_0250.imap	submit file	Re-baselined HST imaps as 250-series for OPUS 2014.2	[opus_71_i.cat', 'hst_acs_0250.imap', 'hst_cos_0250.imap', 'hst_nicmos_0250.imap', 'hst_stis_0250.imap', 'hst_wfc3_0250.imap', 'hst_wfc2_0250.imap']
				[opus_70_i.cat', 'hst_acs_atdtab_0250.rmap', 'hst_acs_biasfile_0250.rmap', 'hst_acs_bpixtab_0250.rmap', 'hst_acs_ccdtab_0250.rmap', 'hst_acs_cthfile_0250.rmap', 'hst_acs_crejtab_0250.rmap', 'hst_acs_d2mfile_0250.rmap', 'hst_acs_darkfile_0250.rmap', 'hst_acs_dgeofile_0250.rmap', 'hst_acs_drkfile_0250.rmap', 'hst_acs_fshfile_0250.rmap', 'hst_acs_idctab_0250.rmap']

The details vary by the type of action, in this case showing the original name of a file prior to submission to CRDS and the assignment of its official name.

8.4 Private Functions

The following functions are restricted to users with accounts on the CRDS website and support the submission of new reference and mapping files and maintenance of the overall site. Private functions are only visible to users who have successfully logged in.

8.4.1 Login and Instrument Locking

Typical batch file submissions automatically generate instrument and pipeline context files, as well as .rmaps. To preclude the possibility of multiple users submitting files from the same instrument at the same time, and possibly creating conflicting rules, users lock instruments when they log in.

When a user logs in, the instrument they've locked and the time remaining on the lock are displayed below the login (now logout) button:

Start Date	Context	Status	Description
2014-08-18	hst_0271.pmap	operational	New ACS BIAS FILES wfc1-1k_gain2_140803_bia.fits wfc1-2k_gain2_140516_bia.fits
2014-08-11	hst_0270.pmap	archived	Delivery of new WFC3/UVIS post-flashed darks.
2014-08-05	hst_0269.pmap	archived	New STIS darks and biases delivered for the new anneal month.
2014-07-15	hst_0268.pmap	archived	Delivery of new WFC3 UVIS post-flashed dark files.

The time displayed is the relative time remaining on the lock reservation, nominally around 4 hours with the current server configuration.

When the user performs an action on the website, their lock timer is reset to its maximum value. As time passes without action, the lock timer counts down. When the lock timer reaches zero, the lock is automatically released and

any on-going file submission is cancelled. Files which have been uploaded for a cancelled submission are left in the upload area.

Other users who attempt to login while an instrument is locked will be denied.

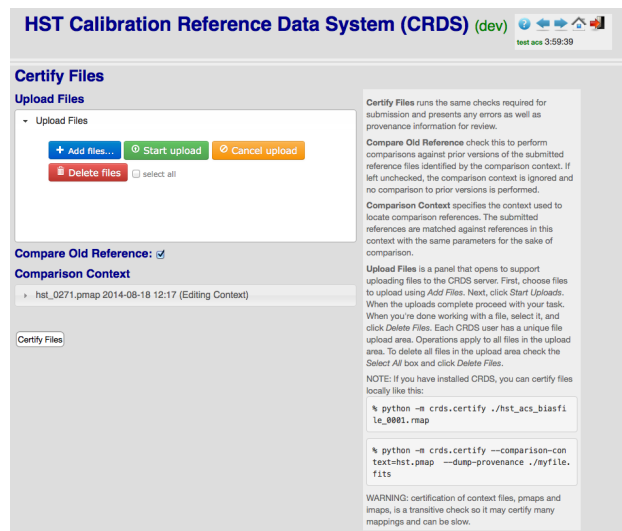
When a file submission is being performed, it must be *confirmed* within the timeout period or the file submission will be cancelled.

Care should be taken with the locking mechanism and file submissions. **DO NOT:**

- Don't login from multiple browsers or sites. The last browser/site you log in from will steal the lock from the original login, cancel any original file submission, and force a logout in the original browser.
- Don't leave the page during an ongoing file submission, wait for it to finish. Opening other browser tabs should be fine.
- Don't attempt to login for more than one instrument at a time. One user is assigned one and only one lock.
- Don't attempt to perform multiple file submissions for the same instrument at the same time. Finish and confirm or cancel each file submission before proceeding with the next.

8.4.2 Certify Files

Certify File runs `crds.certify` on the files in the ingest directory.



If the certified file is a reference table, the specified context is used to locate a comparison file.

8.4.3 Mark Files Bad

Mark Files Bad supports marking a file as scientifically invalid and also supports reversing the decision and marking it good once more.

The CRDS procedure for marking files bad requires three steps:

1. Create a clean context which does not contain any prospective bad files.
2. Make the clean context operational using Set Context.
3. Mark the prospective bad files actually bad using Mark Bad Files.

HST Calibration Reference Data System (CRDS) (dev) test acs 3:59:48

Mark Files Bad

Files:

Mark as: ☐ Bad ☐ OK

Reason:

Mark Files Bad Marks files as "bad" or "ok".
Marking a file bad means that it is scientifically invalid or likely to produce invalid results.
Contexts marked bad or containing bad files cannot be set as the operational default.
Files which are in the operational context cannot be marked bad.
To mark a file bad, first generate an alternate context which omits the bad file and make it operational. The bad file will transition to a non-operational state. Non-operational files can be marked bad and used under historical contexts.

Following this procedure maintains the invariant that the operational context contains no known bad files.

Marking a rules file (mapping) as bad implicitly marks all the files which refer to it as bad. Hence, marking a .rmap as bad will make any .imap which refers to it bad as well, and will also taint all .pmaps which refer to the bad .imaps. Whenever a rules file is marked bad, a warning is issued when the containing context is used.

Marking a reference file as bad is a more precise technique which invalidates only that reference in every context that includes it. Warnings are issued related to the bad reference only when the reference is actually recommended by CRDS.

8.4.4 Set Context

Set Context enables setting the operational and edit contexts.

HST Calibration Reference Data System (CRDS) (dev) test acs 3:59:39

Set Context

Select:

Context Type	Current Value
operational	hst_0271.pmap 2014-08-18 12:17

Set To:

hst_0271.pmap 2014-08-18 12:17 (Editing Context)

Reason for Change:

Set Context is typically used to define the pipeline mapping which will be used in operations. Set Context can also be used to modify the context which will be used as the basis of the next set of rules.
Operational Context Default pipeline mapping which will be used operationally to determine best references. The operational context cannot contain blacklisted references.
Edit Context Default starting point for editing/deriving new mappings. Unless overridden, new file submissions which generate rules will use this as the starting context and create a modified version of it.
Select: Choose the type of default to modify, operational or edit. This also shows the current default of the selected type.
Set To: The default context of the chosen type will be set to this value.

CRDS enables contexts to be pre-positioned before their adoption as the default for processing by the pipeline. Only by using Set Context will an available context become the default for processing.

Setting the operational context makes the specified context the default for processing coordinated by this server. Setting the operational context creates a new entry at the top of the Context History.

Setting the edit context makes the specified context the default starting point for future contexts created during file submission.

8.4.5 Batch Submit References

Batch Submit References is intended to handle the majority of CRDS reference submissions with a high degree of automation. This page accepts a number of reference files and metadata which is applied to all of them. The specified reference files are checked on the server using `crds.certify` and if they pass are submitted to CRDS. All of the submitted references must be of the same reference type, i.e. controlled by the same `.rmap` file. Tabular reference files are checked with respect to the derivation context by `crds.certify`.

Upload Files

The first task involved with *Batch Submit References* is transferring the submitted files to the server. For CRDS build-2, there are two approaches for getting files on the server, web based and shell based. Both approaches involve transferring files to an ingest directory in the CRDS filestore. Each CRDS user will have their own ingest directory. Initially the only user is “test”. This section applies equally to all of the file submission pages that have an *Upload Files* accordion.

Web Approach

On the file submission pages, the *Upload Files* accordion opens to support uploading submitted files to a user’s CRDS ingest directory via the browser.

Uploading files is accomplished by:

- Opening the accordion panel by clicking on it.
- Add files to the upload list by clicking on the *Add Files...* button. Alternately for modern browsers (Chrome) drag-and-drop files from your desktop to the upload accordion.

- Click *Start Upload* to initiate the file transfer. You should see a progress bar(s) showing the status of the upload(s). When the upload successfully completes the buttons will change to *delete*.
- Click *Delete* for any file added by mistake or for failed uploads.
- Click *Cancel Upload* to abort a file transfer during the upload.
- Close the accordion panel by clicking on it.

IMPORTANT Just adding files to the file list does not upload them. You must click *Start upload* to initiate the file transfer. In the screenshot above, the file with the *delete* button next to it is already on the server in the ingest directory. The files with *start* and *cancel* buttons next to them have only been declared as candidates for upload. To finish uploading all 3 files, check *select all* and click *Start upload*.

Shell Approach

In the shell approach a user must login to UNIX (in some fashion) and transfer files into their CRDS ingest directory manually. The nominal approach for doing this is to use the cp or scp commands. For instance, from my home, having already set up ssh and scp access, I might say:

```
% scp /this_delivery/*.fits dmsinsvm.stsci.edu:/ifs/crds/hst/test/server_files/ingest/mcmaster
```

to copy references into my ingest directory *as-if* I had uploaded them through the uploads panel.

Abstractly this is:

```
% scp <submitted reference files...> <host>:/ifs/crds/hst/<pipeline>/server_files/ingest<crds_user>
```

where pipeline is ‘test’ or ‘ops’.

The submitted reference files should now be in the ingest directory for *HST* test server user *mcmaster*. Once the files are in the ingest directory, the CRDS web server will behave as if they had been uploaded through web interface. Refreshing the file submission web page should make manually copied files show up in the *Upload Files* accordion.

The purpose of using cp or scp is to improve the efficiency and reliability of the file transfers should those become an issue. Telecommuters working offsite by VPN would face a situation where submitted files are downloaded to their home computer via VPN and then uploaded to the CRDS server via their browser.

Files transferred to the ingest directory via shell should still be removeable using the *Upload Files* delete buttons.

Derive From Context

The specified context is used as the starting point for new automatically generated context files and also determines any predecessors of the submitted references for comparison during certification. If all the submitted reference files pass certification, a new .rmap, .imap, and .pmap are generated automatically to refer to the newly entered references. Based on their header parameters, references are automatically assigned to appropriate match locations in the .rmap file.

Derive From Context

▼ hst_0002.pmap 2012-09-30 09:33 (Editing Context)

☒ Editing

hst_0002.pmap 2012-09-30 09:33
or more recent

☐ Operational

hst.pmap 2012-09-29 13:00
or more recent

☐ Recent

hst.pmap 2012-09-29 13:00 ▼

☐ User Specified

e.g. hst_0001.pmap

There are two special contexts in CRDS which are tracked:

Edit Context

Edit Context is the default context used for editing. Whenever a new .pmap is created or added, it becomes the editing context from which other .pmaps are derived by default.

Operational Context

Operational Context is the .pmap which is nominally in use by the pipeline. Generally speaking, multiple contexts might be added to CRDS as the Edit Context long before they become operational.

Recent

Recent lists a number of recently added contexts based on delivery time.

User Specified

Any valid CRDS context can be typed in directly as User Specified.

Auto Rename

Normally files uploaded to CRDS will be assigned new unique names. During side-by-side testing with CDBS, *Auto Rename* can be deselected so that new files added to CRDS retain their CDBS names for easier comparison. The CRDS database remembers both the name of the file the submitter uploaded as well as the new unique name.

Compare Old Reference

When checked CRDS will certify incoming tabular references against the files they replace with respect to the derivation context. For other references this input is irrelevant and ignored.

Results

Batch Reference Submit Results

Starting Context

hst_0002.pmap

Generated New Mappings

hst_0003.pmap
hst_cos_0003.imap
hst_cos_deadtab_0003.rmap

Certify Results

› s7g1700ql_dead.fits -->	hst_cos_deadtab_0004.fits	OK
› s7g1700gl_dead.fits -->	hst_cos_deadtab_0005.fits	OK
› s7g1700gm_dead.fits -->	hst_cos_deadtab_0006.fits	OK

Actions on hst_cos_deadtab_0002.rmap

› Rmap Logical Diffs
› Rmap Text Diffs

Confirm or Abort Submission

The results page lists the following items:

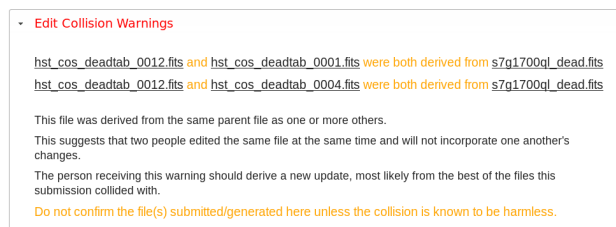
- *Starting Context* is the context this submission derove from.
- *Generated New Mappings* lists the new mapping files which provide the generated context for using the submitted references.

- *Actions on Rmap* provides two accordions showing how the rmap controlling the submitted references was modified. The logical differences accordion has a table of actions, either *insert* for completely new files or *replace* for files which replaced an existing file. The text differences are essentially output from UNIX *diff* for the old and new rmaps.
- *Certify Results* has an accordion panel for each submitted reference file which contains the results from crds.certify. The submitted name of each file is listed first, followed by any official name of the file assigned by CRDS. The status of the certification can be “OK” or “Warnings”. Warnings should be reviewed by opening the accordion panel.

IMPORTANT The results page only indicates the files which will be added to CRDS if the submission is *confirmed*. Prior to confirmation of the submission, neither the submitted references nor the generated mappings are officially in CRDS. Do not *leave the confirmation page* prior to confirming.

Collisions

Under some circumstances, a *Collision Warning* accordion will be present. It should be carefully examined to ensure that overlapping edits of the same context file have not occurred. Overlaps can be resolved by cancelling the current submission and re-doing it, or by accepting the current submission and manually correcting the mappings involved. Failure to correctly resolve a collision will most likely result in one of two sets of conflicting changes being lost.



Collision tracking for CRDS mappings files is done based upon header fields, nominally the *name* and *derived_from* fields. These fields are automatically updated when mappings are submitted or generated.

Collision tracking for reference files is currently filename based. The submitted name of a reference file is assumed to be the same as the file it was derived from. This fits a work-flow where a reference is first downloaded from CRDS, modified under the same name, and re-uploaded. Nominally, submitted files are automatically re-named.

Confirm or Discard

If everything looks good the last step is to click the *Confirm* button. Clicking the Confirm button finalizes the submission process, submits the files for archive pickup, and makes them a permanent part of CRDS visible in the database browser and potentially redistributable. A confirmed submission cannot be revoked, but neither will it go into use until the pipeline or a user explicitly requests it.

Discarding a batch submission based on warnings or bad rmap modifications removes the submission from CRDS. In particular temporary database records and file copies are removed.

Following any CRDS pipeline mapping submission, the default *edit* context is updated to that pipeline mapping making it the default starting point for future submissions.

8.4.6 Submit References

Submit References provides a lower level interface for submitting a list of references. No mappings are generated to refer to the submitted files. Submitted references must still pass through crds.certify.

HST Calibration Reference Data System (CRDS) (dev)

test acs 3:57:54

Submit Reference Files test acs

Upload Files

Change Level: SEVERE

Creator:

Description:

Compare Old Reference: ☒

Comparison Context:

hst_0271.pmap 2014-08-18 12:17 (Editing Context)

Auto-Rename: ☐

Submit Files

Submit Reference Files permanently enters files into CRDS and the archive.

Upload Files is a panel that opens to support uploading files to the CRDS server. First, choose files to upload using Add Files. Next, click Start Uploads. When the uploads complete proceed with your task. When you're done working with a file, select it, and click Delete Files. Each CRDS user has a unique file upload area. Operations apply to all files in the upload area. To delete all files in the upload area check the Select All box and click Delete Files.

Compare Old Reference check this to perform comparisons against prior versions of the submitted reference files identified by the comparison context. If left unchecked, the comparison context is ignored and no comparison to prior versions is performed.

Comparison Context specifies the context used to locate comparison references. The submitted references are matched against references in this context with the same parameters for the sake of comparison.

Change Level The degree to which the new files are expected to impact science results.

Description information about what's new in this file and what the expected impacts are.

Creator author of the contents or changes to the file, not necessarily the submitter. The submitter is known based on login information.

8.4.7 Submit Mappings

Submit Mappings provides a basic interface for submitting a list of mapping files which don't have to be related. This can be used to submit context files which refer to files from *Submit References* and with fewer restrictions on allowable changes. Typically only .rmaps are submitted this way. Mappings submitted this way must also pass through crds.certify.

HST Calibration Reference Data System (CRDS) (dev)

test acs 3:59:55

Submit Mapping Files test acs

Upload Files

Change Level: SEVERE

Creator:

Description:

Generate Contexts: ☒

Derive From Context:

hst_0271.pmap 2014-08-18 12:17 (Editing Context)

Auto-Rename: ☐

Submit Files

Submit Mapping Files permanently enters files into CRDS and the archive.

Upload Files is a panel that opens to support uploading files to the CRDS server. First, choose files to upload using Add Files. Next, click Start Uploads. When the uploads complete proceed with your task. When you're done working with a file, select it, and click Delete Files. Each CRDS user has a unique file upload area. Operations apply to all files in the upload area. To delete all files in the upload area check the Select All box and click Delete Files.

Change Level The degree to which the new files are expected to impact science results.

Description information about what's new in this file and what the expected impacts are.

Creator author of the contents or changes to the file, not necessarily the submitter. The submitter is known based on login information.

Generate Contexts Automatically generate a new .pmap and .jmap utilizing submitted rmaps. Ignored for submitted .pmaps or .jmaps.

Derive From Context Generate new .rmaps and .pmaps by inserting the submitted .rmaps into this context to derive a new one.

WEB SERVICES

The CRDS servers support a JSONRPC based service mechanism which enables remote users to make calls to the CRDS server without installing the CRDS Python based client library. See <http://json-rpc.org/wiki/specification> for more details on the JSONRPC protocol.

9.1 Supported Methods

9.1.1 `get_default_context(observatory)`

get_default_context returns the name of the pipeline mapping which is currently in use by default in the operational pipeline, e.g. 'jwst_0001.pmap'. `get_default_context` is called with a single parameter, *observatory*, which can be 'hst' or 'jwst'.

9.1.2 `get_best_references(context, header, reftypes)`

get_best_references matches a set of parameters *header* against the lookup rules specified by the pipeline mapping *context* to return a mapping of type names onto recommended reference file names.

A suitable *context* string can be obtained from `get_default_context()` above, although any archived CRDS context file can be specified.

The *header* parameter of `get_best_references` is nominally a JSON object which maps CRDS parkey names onto dataset file header values. CRDS parkey names can be located by browsing reference mappings (.rmap's) and looking at the *parkey* header parameter of the rmap.

For JWST, the rmap parkeys (matching parameter names) are currently specified as JWST stpipe data model dotted identifiers. Example JSON for the `get_best_references` *header* parameter for JWST is:

```
{ "meta.instrument.type": "fgs",  
  "meta.instrument.detector": "fgs1",  
  "meta.instrument.filter": "any" }
```

For JWST, it is also possible to use the equivalent FITS header keyword, as defined by the data model schema, to determine best references:

```
{ "instrume": "fgs",  
  "detector": "fgs1",  
  "filter": "any" }
```

For HST, GEIS or FITS header keyword names are supported.

reftypes should be a json array of strings, each naming a single desired reference type. If *reftypes* is passed as null, recommended references for all reference types are returned. Reference types which are defined for an instrument but which are not applicable to the mode defined by *header* are returned with the value *NOT FOUND n/a*.

Example JSON for *reftypes* might be:

```
["amplifier", "mask"]
```

9.2 JSONRPC URL

The base URL used for making CRDS JSONRPC method calls is essentially */json/*. All further information, including the method name and the parameters, are POSTed using a JSON serialization scheme. Example absolute server URLs are:

9.2.1 JWST

```
http://jwst-crds.stsci.edu/json/
```

9.2.2 HST

```
http://hst-crds.stsci.edu/json/
```

9.3 JSONRPC Request

An example CRDS service request can be demonstrated in a language agnostic way using the UNIX command line utility *curl*:

```
% curl -i -X POST -d '{"jsonrpc": "1.0", "method": "get_default_context", "params": ["jwst"], "id": 1}'
HTTP/1.1 200 OK
Date: Fri, 12 Oct 2012 17:29:46 GMT
Server: Apache/2.2.3 (Red Hat) mod_python/3.3.1 Python/2.7.2
Vary: Cookie
Content-Type: application/json-rpc
Connection: close
Transfer-Encoding: chunked
```

The *jsonrpc* attribute is used to specify the version of the JSONRPC standard being used, currently 1.0 for CRDS.

The *method* attribute specifies the name of the service being called.

The *params* attribute specifies a JSON array of parameters which are passed positionally to the CRDS method.

The *id* can be used to associate calls with their responses in asynchronous environments.

9.4 JSONRPC Response

The response returned by the server for the above request is the following JSON:

```
{"error": null, "jsonrpc": "1.0", "id": 1, "result": "jwst_0000.pmap"}
```

9.5 Error Handling

Because `get_best_references` determines references for a list of types, lookup errors are reported by setting the value of a reference type to “NOT FOUND ” + `error_message`. A value of “NOT FOUND n/a” indicates that CRDS determined that a particular reference type does not apply to the given parameter set.

Fatal errors are handled by setting the error attribute of the result object to an error object. Inspect the `result.error.message` attribute to get descriptive text about the error.

9.6 JSONRPC Demo Page

The CRDS servers support demoing the JSONRPC services and calling them interactively by visiting the URL `.../json/browse/`. The resulting page is shown here:

An example dialog for `get_best_references` from the CRDS jsonrpc demo page is shown here with FITS parkey names:

```
>>> jsonrpc.get_best_references("jwst_0000.pmap", {'INSTRUME':'FGS','DETECTOR':'FGS1', 'FILTER':'ANY'})
Requesting ->
{"id":"jsonrpc", "params":["jwst_0000.pmap", {"INSTRUME":"FGS", "DETECTOR":"FGS1", "FILTER":"ANY"}], n
Deferred(12, unfired)
Got ->
{"error": null, "jsonrpc": "1.0", "id": "jsonrpc", "result": {"linearity": "jwst_fgs_linearity_0000.l
```

And the same query is here with JWST data model parkey names:

```
>>> jsonrpc.get_best_references("jwst_0000.pmap", {'META.INSTRUMENT.TYPE':'FGS','META.INSTRUMENT.DETE
Requesting ->
{"id":"jsonrpc", "params":["jwst_0000.pmap", {"META.INSTRUMENT.TYPE":"FGS", "META.INSTRUMENT.DETECTOR
Deferred(14, unfired)
Got ->
{"error": null, "jsonrpc": "1.0", "id": "jsonrpc", "result": {"linearity": "jwst_fgs_linearity_0000.l
```

NOTE: An apparent bug in the demo interpreter makes it impossible to pass the `get_best_references` *reftypes* parameter as an array of strings. In the current demo *reftypes* can only be specified as null.

JSON-RPC Browser

Methods

get_reference_names

get_best_references

get_file_chunk

get_url

list_mappings

get_server_info

file_exists

get_file_info

get_default_context

get_mapping_names

Console

```
{ "error": { "executable": "/usr/local/bin/python2.7", "code": 500, "name":
"OtherError", "message": "OtherError: Missing 'META.INSTRUMENT.TYPE' keyword in header",
"data": null, "stack": "Traceback (most recent call last):\n File \"/grp/crds/jwst/webserver
/python/lib/python/django_json_rpc-0.6.2-py2.7.egg/jsonrpc/site.py", line 152, in
response_dict\n R = apply_version[version](method, request, D['params'])\n File \"/grp
/crds/jwst/webserver/python/lib/python/django_json_rpc-0.6.2-py2.7.egg/jsonrpc/site.py", line
125, in <lambda>\n '1.0': lambda f, r, p: f(r, *p)}\n File \"/grp/crds/jwst/webserver/python
/lib/python/crds/server/jsonapi/views.py", line 141, in get_best_references\n return
rmap.get_best_references(context, conditioned, include=reftypes)\n File \"/grp/crds/jwst
/webserver/python/lib/python/crds/rmap.py", line 1057, in get_best_references\n return
ctx.get_best_references(header, include=include)\n File \"/grp/crds/jwst/webserver/python
/lib/python/crds/rmap.py", line 501, in get_best_references\n instrument =
self.get_instrument(header)\n File \"/grp/crds/jwst/webserver/python/lib/python/crds/rmap.py",
line 583, in get_instrument\n raise crds.CrdsError("\nMissing '%s' keyword in header\n" %
self.instrument_key)\nCrdsError: Missing 'META.INSTRUMENT.TYPE' keyword in header\n"},
"jsonrpc": "1.0", "id": null, "result": null}
>>> jsonrpc.get_best_references("jwst_0000.pmap",
{'META.INSTRUMENT.TYPE': 'FGS', 'META.INSTRUMENT.DETECTOR': 'FGS1',
'META.INSTRUMENT.FILTER': 'ANY'}, null)
Requesting ->
{"id": "jsonrpc", "params": [{"jwst_0000.pmap", {"META.INSTRUMENT.TYPE": "FGS",
"META.INSTRUMENT.DETECTOR": "FGS1", "META.INSTRUMENT.FILTER": "ANY"}, null],
"method": "get_best_references", "jsonrpc": "1.0"}
Deferred(4, unfired)
Got ->
{"error": null, "jsonrpc": "1.0", "id": "jsonrpc", "result": {"linearity":
"jwst_fgs_linearity_0000.fits", "amplifier": "jwst_fgs_amplifier_0000.fits", "mask":
"jwst_fgs_mask_0000.fits"}}
```

- JSON-RPC methods are exposed through the `jsonrpc` global.
- Call a method just as you would a javascript one: `jsonrpc.jsonrpc.test('arg1')`
- Calling a method returns a `Deferred` (see the [MochiKit docs](#)). As soon as the deferred returns the result is available by the global variables `json_result` or `json_error`
- Use `dir` to explore the namespace: `dir(jsonrpc)`

CRDS DATABASE ACCESS

10.1 JSON RPC Access

CRDS supports JSON RPC access to the CRDS catalog via the `crds.client` API.

10.1.1 Metadata for a single file

The JSON RPC call `get_file_info()` will return the available info for a single reference or mapping file from the specified observatory:

```
def get_file_info(observatory, filename):
    """Return a dictionary of CRDS information about `filename`."""

>>> from crds.client import api

>>> api.get_file_info("hst", file="lcb12060j_drk.fits")
{'activation_date': '2001-12-14 20:47:00',
 'blacklisted': 'false',
 'change_level': 'severe',
 'delivery_date': '2013-07-10 11:26:23',
 'derived_from': 'none',
 'filekind': 'darkfile',
 'instrument': 'acs',
 'name': 'lcb12060j_drk.fits',
 'observatory': 'hst',
 'pedigree': 'ground',
 'rejected': 'false',
 'shasum': '56cfd1107bda5d82cb49a301a50edb45cb64ded6',
 'size': '10549440',
 'state': 'operational',
 'type': 'reference',
 'useafter_date': '1992-01-01 00:00:00'}
```

10.1.2 Metadata for several / all files

The JSON RPC call `get_file_info_map()` will return the info for multiple (or all) files and the specified (or all) fields as a dictionary of dictionaries mapping filename onto info:

```
def get_file_info_map(observatory, files=None, fields=None):
    """Return the info { filename : { info } } on `files` of `observatory`.
    `fields` can be used to limit info returned to specified keys.
    """
```

```
% setenv CRDS_SERVER_URL https://hst-crds.stsci.edu

>>> from crds.client import api

>>> api.get_file_info_map("hst", ["lcb12060j_drk.fits", "n3o1022fj_drk.fits"], fields=["state", "size"])
{'lcb12060j_drk.fits': {'shasum': '56cfd1107bda5d82cb49a301a50edb45cb64ded6',
  'size': '10549440',
  'state': 'operational'},
'n3o1022fj_drk.fits': {'shasum': 'cecf11300015df8f39913b638138d8c67de77a02',
  'size': '10526400',
  'state': 'operational'}}
```

If files is specified as *None*, info on all files is returned.

If fields are specified as *None*, info on all available fields is returned.

10.2 Download CRDS catalog for SQLite queries

The CRDS catalog stores metadata about references not captured in the .rmap files. It also contains the history of CRDS context use, the effective dates at which particular contexts were operational in the pipeline.

You can download a SQLite-3 snapshot of the CRDS catalog like this:

```
% setenv CRDS_SERVER_URL https://hst-crds.stsci.edu
% setenv CRDS_PATH /home/jmiller/crds_cache
% python -m crds.sync --fetch-sqlite-db
CRDS : INFO      SQLite database file downloaded to: /home/jmiller/crds_cache/config/hst/crds_db.sqlite3
```

will snapshot the current CRDS catalog on the CRDS server and download it to your local CRDS cache as a SQLite3 database file. The SQLite database can typically be accessed like this:

```
% sqlite3 /home/jmiller/crds_cache_dev/config/hst/crds_db.sqlite3

sqlite> .tables
crds_hst_catalog      crds_hst_context_history

sqlite> .mode tabs
sqlite> .headers on
sqlite> select * from crds_hst_context_history where state="operational" limit 1;
id      name      start_date      context      state      description
2       2013-07-02 15:44:53      hst.pmap      operational      set by system
\\.\\.\\.\\.
```

The CRDS catalog contains the following meta-data:

Catalog Fields	type	description
name	str	CRDS filename
uploaded_as	str	Name of file at time of upload / generation
state	str	uploaded, delivered, submitted, archiving, archived, archiving-failed, bad
blacklisted	bool/int	True/1 == this mapping, and all mappings referring to it, are invalid.
rejected	bool/int	True/1 == this file is considered scientifically invalid
replaced_by_filename	str	Succeeding reference file in chain of contexts. Weakly defined.
instrument	str	instrument name file applies to
filekind	str	reference type. For HST, also keyword name for dataset headers
type	str	reference or mapping
description	str	description given at time of delivery
comment	str	COMMENT from reference file
aperture	str	APERTURE from reference file
derived_from	str	Name of mapping this one was derived from
sha1sum	str	sha1sum of file to verify file integrity
size	int	length of file in bytes
creator_name	str	author of reference or mapping file
deliverer_user	str	person who submitted the reference or mapping to CRDS
deliverer_email	str	e-mail of person who submitted reference

NOTE: Reference file assignment criteria are encoded in the CRDS rules / mappings and displayed as tables on the web site context display. See also `crds.matches` for information on displaying matching criteria based on `rmaps` at the command line.