

Flight Rule Evaluation Sequencing Helper (FRESH) Adapter's Guide

For adapters and maintainers of FRESH

- [Flight Rule Evaluation Sequencing Helper \(FRESH\) Adapter's Guide](#)
 - [For adapters and maintainers of FRESH](#)
 - [Overview](#)
 - [Design Concepts](#)
 - [Configuration](#)
 - [Creating or updating a mission specification](#)
 - [Creating a configuration file](#)
 - [Updating the default configuration file](#)
 - [Inputs and Outputs](#)
 - [The SeqDict Class](#)
 - [SeqDict Properties](#)
 - [SeqStep Properties](#)
 - [SeqTimeTag Properties](#)
 - [SeqArg Properties](#)
 - [SeqDict Enums](#)
 - [Writing an Input Method](#)
 - [FRCheckInfo and FRResults Objects](#)
 - [FRCheckInfo Properties:](#)
 - [FRResult\(NamedTuple\):](#)
 - [FRState and FRCriticality Enums](#)
 - [FRESH Report Output](#)
 - [Calling FRESH in other contexts](#)
 - [CSV Rules](#)
 - [Creating, updating, or deleting a CSV rule](#)
 - [Command Existence Rule Specification](#)
 - [Command Argument Rule Specification](#)
 - [Command Timing Rule Specification](#)
 - [Creating a new CSV rule type](#)
 - [Python Rules](#)
 - [Creating a new Python rule](#)
 - [Updating or deleting a Python rule](#)
 - [Testing](#)
 - [Unit Testing with Pytest](#)
 - [Test Utils](#)
 - [CLIArgs class](#)
 - [CLIArgs.default_for_test method](#)
 - [CLIArgs.custom_for_test method](#)
 - [CLIArgs.run_results_only_test method](#)
 - [CLIArgs.run_simple_test method](#)
 - [CLIArgs.run_test method](#)
 - [CLIArgs.get_step method](#)
 - [CLIArgs.get_arg method](#)
 - [CLIArgs.match_to_steps method](#)
 - [CLIArgs.assert_all_steps_have_state method](#)
 - [CLIArgs.match_steps_by_state method](#)
 - [Packaging and Delivery](#)
 - [Building the FRESH Package](#)
 - [Installing FRESH Locally](#)

- [How to use the build_install_test script](#)

Overview

FRESH is a limited flight rule evaluation tool for use on SEQ JSON files. It runs on a single SEQ JSON file, evaluating a number of flight rules as defined by the configuration of FRESH at runtime, and outputs a JSON file called a FRESH report. The configuration points to CSV files which define the flight rules to be checked. The FRESH report provides the status of every rule in the configuration and in the FRESH Python flight rules code, whether or not it was checked, and detailed results of each individual check.

This guide is designed to be an in-depth review of the code. For a general overview of FRESH and how to use it, please see the User's Guide.

Design Concepts

There are three driving design concepts for FRESH:

1. Modularity of inputs and outputs, which use common interfaces to the "core" library.
2. Reduce false negatives as much as possible, even if the number of false positives increases.
3. Allow for easy and fast authoring of simple rules, rather than covering every possible rule use case.

Concept 1 drives the structure of FRESH. Flight rules should be self-contained checks which allow them to be swapped in and out easily, or updated without breaking other rules. The SeqDict class functions as the common interface to all input types. IO Methods can be written for any desired input type which can support the information required by the SeqDict class and added to a FRESH adaptation. FRESH writes output to a JSON file which can be used by other tools to create a visual report.

Concept 2 drives the strategy for marking rules as flagged or violated. When there is uncertainty about the evaluation of a rule, or if there is information needed to fully evaluate a rule that FRESH does not have access to, FRESH will not mark a rule as PASSED. The exact implementation of this philosophy can vary from one flight rule to another.

Concept 3 drives the implementation of the three CSV rule definition file types, which are defined in more detail in following sections. The CSVs combined with the configuration allows users to add more rules (or remove rules) without requiring the software to be redelivered. There are limitations to the type of rules that can be defined in these CSV files, so FRESH also provides a Python class interface for defining custom rule checks. The custom rule checks cannot be swapped in or out at runtime.

When adapting FRESH, these three design concepts should be kept in mind, and only broken if there is a clear need, a use case, and no alternative solution. As more capabilities are added to FRESH the ease of maintainability naturally decreases, so adapters should be careful to weigh complicated additions against the future maintainance costs. Whenever possible, new features should use the interfaces already designed for the purpose, such as SeqDict and the report JSON.

Configuration

A FRESH adaption is configured in two ways: using the MISSION variable in `mission.py` to set the mission configuration, and using a JSON configuration at runtime.

The mission configuration is only used by developers, and sets various parts of the code and build process to the correct values for a given mission. It cannot be changed at runtime, and should only be overridden by core tests and only when absolutely necessary. The mission is set using the `MISSION` variable in the [mission.py](#) file, which is used by the methods in [mission_config.py](#) to set all mission-specific values.

The JSON runtime configuration, usually shortened to the config file, is used to specify information that FRESH needs to set up a run of the tool. This includes the locations of CSV rule files, the location of a command dictionary, and any other information that is required by an adaptation.

Creating or updating a mission specification

To set the mission that FRESH is running, the `MISSION` variable in [mission.py](#) should be set to the mission identifier. This identifier should be used throughout FRESH to designate mission specific information, such as the flight rule folders, test folders, and custom test marks. Then, the methods in `mission_config.py` [mission_config.py](#) should be updated to include a value for the mission identifier. The methods in `mission_config.py` are called in various parts of the FRESH code.

Any new mission specific values should be added to `mission_config.py` as a method following this pattern, so future missions or updates have a centralized location for updating or auditing these values. All methods for the mission configuration should use the `unknown_mission` method to handle any mission identifiers that do not have an if case for that method. The `unknown_mission` method raises a `NotImplemented` error with a custom error message including the mission identifier, and a short description of the missing information which should be passed to the method.

Creating a configuration file

The config file is a JSON file. There are a few required keys:

- `command_existence_rule_files`: a list of file paths pointing to command existence rule files.
- `command_argument_rule_files`: a list of file paths pointing to command argument rule files.
- `command_timing_rule_files`: a list of file paths pointing to command timing rule files.
- `command_dictionary_path`: a single directory pointing to an XML command dictionary file.

Absolute or relative paths can be used in the config file. All relative paths in the config file are interpreted relative to the config file location, not the install location of FRESH or the call location at runtime. This allows FRESH to point to a config file from anywhere. All keys which expect a list treat an empty list as a valid option.

Configuration files do not strictly require that only the above keys exist, so additional keys can be added with no harm. The information in the config file, as well as the directory location of the config file itself, are accessible by all flight rules. This means that any of the information in the config file can be used in a flight rule check, including additional information which may be added by the adapter. If a flight rule does require additional config information, that flight rule is responsible for raising an exception if the required key is missing.

Updating the default configuration file

A delivered FRESH package should include a default config file, to allow users to run FRESH without writing their own configuration. It is intended to be a configuration including all delivered flight rules for that mission, but this is not a requirement if a mission decides otherwise. Default configuration files should be included in the [fresh/config folder](#). Multiple files can be present in the config folder, but only one file can be assigned as the default for a mission. Selecting or changing a default configuration file is done in `mission_config.py`, which contains a method to return the filename of the default config for a given mission identifier.

The default configuration file is included in the python package with the rest of FRESH, so updating the default config requires rebuilding the FRESH package. Any updates made to the default configuration file will be present in the build.

Inputs and Outputs

The SeqDict Class

The SeqDict class is the format in which sequence (or other input) information is passed through FRESH for flight rule evaluation. By using a custom structured object rather than a dictionary with implied structure, FRESH can restrict knowledge of the input format to an IO method which outputs a SeqDict for the rest of the program to use. The IO methods can then be switched out as necessary, or be chained such that FRESH calls an IO method, which looks at the input it is given and selects the appropriate method to turn it into a SeqDict, and returns the result. This allows current and future users to swap out the input of FRESH without needing to rewrite large portions of the code.

SeqDict does make two key assumptions about the information that is being checked. First, the information can be roughly condensed into a series of timed "steps". Second, the steps that are relevant to most checks will contain an identifier, often called a "stem". These assumptions come from FRESH being developed to check flight rule compliance on sequences, but theoretically any information that can provide timed "steps" with identifiers could be fit into this structure. If the input cannot meet the spirit of these assumptions, FRESH is probably not the right tool to perform the checks.

The structure of a SeqDict object and its sub-objects is as follows:

SeqDict Properties

`id`: A string identifier for the sequence or input file.

`steps`: A list of SeqStep objects representing the steps in the sequence.

SeqStep Properties

`args`: A list of SeqArg objects, each representing an argument for the step. May be empty.

`stem`: A string identifier for the step to be taken. For a standard command step, this would be the string representation of the command stem, but this may be different if it is not a command step. The `stem` field is what is used by FRESH to determine if a flight rule applies to a given step, and should be filled out accordingly.

`time`: A SeqTimeTag object representing the execution time of the step.

`step_type`: A SeqStepType enum value indicating the type of the step.

SeqTimeTag Properties

`tag`: A string representing the time value of the time tag. Can be `None` for types that execute immediately upon receipt, or as fast as possible from the previous command.

`time_type`: A SeqTimeType enum value indicating the type of the time tag.

SeqArg Properties

`arg_type`: A SeqArgType enum value indicating the type of the argument.

`value`: The value of the argument. No typing, as this will vary based on the `arg_type`.

`name`: The name of the argument. It is the responsibility of the IO method to provide this information at the creation of the SeqDict. The name may need to be inferred by the IO method from the ordering of the arguments in the input file, but this is not sufficient information for the core FRESH to perform flight rule checks.

SeqDict Enums

SeqDict uses several custom enumerations to streamline value checking where there may be multiple valid string interpretations, such as `COMMAND`, `command`, `cmd`, etc. Most enums provide a `from_string` method which checks against a list of valid strings and returns the appropriate enum. Additional values can be added to these lists in `from_string` methods as necessary.

- SeqStepType is an enum representing the type of a SeqStep.
- SeqTimeType is an enum representing the type of a SeqTimeTag. Not all values will be appropriate or valid for all missions, but checking the validity should be done by either the IO method or flight rules that are provided by the mission, not the SeqDict or SeqTimeType class.
- SeqArgType is an enum representing the type of an SeqArg argument for a SeqStep. Not all values will be appropriate or valid for all missions or all input types, but checking the validity should be done by either the IO method or flight rules provided by the mission. Additional values can and should be added if needed. Numeric values are spaced with the intention of keeping similar argument types grouped; for example, all unsigned integer types are given a 2X value, all float types are given a 4X value, etc.

For the most specific and up to date values for the enums, please reference the code in [fresh/seqdict.py](#).

Writing an Input Method

FRESH does not impose many restrictions on writing a class to handle sequence inputs. IO handlers for different data types should be added to the `fresh/fresh_io` folder, but FRESH will not automatically pick them up the way flight rules are found. A pointer to a method which returns a SeqDict object should be provided by the mission configuration in `fresh/mission_config.py`, which will be used by FRESH to handle input when running with that mission identifier. FRESH expects this method to return only the SeqDict and take only the input file path and the config dictionary, although additional information may be added to the config dictionary for the method to use. For example, if a path to a command dictionary file is provided in the config dictionary, the IO method can use that path to read the command dictionary and look up any additional information it requires.

FRCheckInfo and FRResults Objects

The results of flight rule checks performed by FRESH are collected in custom objects. The FRCheckInfo class acts as a summary of the state of a flight rule check, while the FRResult captures the result of a single check performed in relation to a defined flight rule ID. Both classes have `to_json_dict` and `to_json_str` methods to facilitate their use in other contexts. The FRCheckInfo class also overrides the `__add__` method (also accessible with `+`) to allow users to either combine two FRCheckInfo objects with the same `flight_rule_id` into a single FRCheckInfo object, or add a single FRResult object to an existing FRCheckInfo object. In either case, only the lowest state (see FRState below) will be preserved.

FRCheckInfo Properties:

`flight_rule_id`: A string unique flight rule identifier. FRCheckInfo objects with the same `flight_rule_id` will be combined by FRESH when producing a report.

`flight_rule_version`: A string version identifier for the flight rule. This is currently tracked manually by implementers, and may be in whatever format is most useful to the mission.

`flight_rule_description`: A string description of the flight rule.

`criticality`: A FRCriticality enum value indicating the criticality level of the flight rule.

`state`: A FRState enum value indicating the end state of this flight rule. It should be equal to or lower than the state of the highest child in `results`, IE a VIOLATED flight rule may have a mix of VIOLATED, FLAGGED, and PASSED states in the `results` objects, but a FLAGGED flight rule should never have a VIOLATED child in the `results` list.

`num_violations`: An integer number of violations for the given flight rule.

`results`: A list of FRResult objects reporting the results of individual checks performed on the sequence with the same `flight_rule_id`. This may be empty for positive (PASSED or higher) results, and should contain at least one result for negative (FLAGGED, VIOLATED) results.

The `flight_rule_description` for CSV rules is currently generated by combining the messages of any individual checks (rows) with a matching `flight_rule_id`. This can lead to repetitive or unwieldy descriptions, but is a known limitation of the current version of the tool.

FRResult(NamedTuple):

`state`: A `FRState` enum value indicating the end state of this check.

`step_number`: The step number most related to this individual check result. If more than one step is relevant to the check, the first step in violation is generally chosen as the step to "pin" the `FRResult` to.

`command_stem`: The command stem of the step in question.

`command_args`: A list of the arguments of the step in question, if any. May be empty if there are no arguments for the step.

`flight_rule_id`: The ID of the flight rule which requested this check.

`message`: A string message of any length provided to the user. Intended to assist the user in correcting or dispositioning the flight rule violation.

By FRESH convention, values of `-1` for `step` and `SEQUENCE` for `command_stem` should be used for individual result checks where a check was performed over the entire sequence, and no steps were found to be relevant to the rule. This should only be used as a last choice if there is no more useful step information to provide.

FRState and FRCriticality Enums

The flight rule results objects use two enums for clarity of coding.

`FRState` is an enum of the valid end states of a flight rule check. The possible values are:

- `VIOLATED` = -10
- `FLAGGED` = -5
- `PASSED` = 0
- `NA` = 1
- `PENDING` = 10

`FRCriticality` is an enum of the criticality types of flight rules, which equates to the sub-sections of the `fr_results` section in the FRESH report. The possible values are:

- `CAT_A` = 1
- `CAT_B` = 2
- `CAT_C` = 3
- `GUIDELINE` = 4

FRESH Report Output

FRESH produces a report which details the results of all flight rule checks it performed on a sequence in a given run. The report is given as a JSON file which is optimized to be read by other tools. Future plans for the tool include a report visualizer to display the report.

The full structure of the report is detailed in the user's guide, but a summary is provided here.

The metadata section includes information about when the report was run and with which version of FRESH.

The summary section is a rollup of the rules with various alert levels, including passed, and the total number of rule checks performed.

The flight rule results section includes all the returned results from the checks performed by FRESH. This is essentially a JSON print out of the information contained in the `FRCheckInfo` objects and their child `FRResult` objects.

The metadata and summary sections are generated by the `report_io.make_json_dict_from_results` method, and thus do not appear in the output of a direct call of FRESH using the `check_frs.check_flight_rules` method. Only the equivalent of the flight rule checks section is present in the list returned from that method.

Calling FRESH in other contexts

The FRESH library can be imported as a Python package and called from contexts other than the command line, such as a Jupyter notebook. There are two recommended entry points for calling FRESH in this way: `check_frs_from_file` or `check_flight_rules`, both of which are in the `fresh/check_frs` module.

The `check_frs_from_file` method is best used by orchestrators or other tools which want to replicate the functionality of calling FRESH from the command line. The method takes the same arguments as the command line with the addition of `io_method`, which should be a method for reading in the given sequence file and returning a SeqDict object. The input file, output file, and config file should all be given as file paths and a full report is written to the output location.

The `check_flight_rules` method is optimized to be called in a Python context. It takes three arguments: a SeqDict object, a dictionary containing configuration information, and an optional flag for running in verbose mode. This allows the caller to create the SeqDict and config however they would like, without relying on file structures. The output of the method is the results of checking the flight rules as a list of `FRCheckInfo` objects. If desired, the resulting list can be transformed using the `report_io` module methods `make_json_dict_from_results`, which returns a dictionary using only base Python objects and includes summary and metadata information, or `write_fresh_json_report`, which writes the output to a FRESH report file.

CSV Rules

Creating, updating, or deleting a CSV rule

CSV rules can be added as simply as adding a line to a CSV file referenced in the config. The specification of the CSV files should match the type of CSV flight rule it is listed as in the config; see the following sections for detailed specifications. CSV flight rules can be updated in the same way, by editing CSV files and removing the previous versions from the config files.

Multiple files can refer to the same flight rule ID, as can multiple lines in the same file. Each line should refer to a single check, so flight rules that require multiple checks are expected to span multiple lines. FRESH will combine all results with the same flight rule ID into a single `FRCheckInfo` with the highest warning level taking precedence.

CSV rules can be removed at runtime by removing the relevant file path from the configuration file. This will remove all rules in a file, so the recommended FRESH pattern is to combine rules in files based on when they may need to be added or removed, or by rule ID.

Command Existence Rule Specification

Command existence rules are the simplest and most straightforward of the supported CSV flight rule types. Given a command stem, the flight rule will alert at the given level any time it sees that command stem. There are only five fields required, all of which are common to all CSV rule types.

Name	Description	Format	Restrictions
FR_ID	The identifier of the flight rule.	Group-Criticality-Number	Criticality must be one of A, B, C. Number is suggested to be a 0-padded four digit number.
FR_Version	The version of the flight rule	A string of any length.	None
Alert_Level	The level of alert (violation) if the flight rule check is found to be noncompliant.	A valid FRESH FRState.	Must be one of VIOLATED, FLAGGED.

Name	Description	Format	Restrictions
Command_Stem	The command stem of the command to be checked by the flight rule.	A valid command stem.	None
Message	A message to be included when the flight rule is triggered and found to be noncompliant.	A string of any length.	None

Command Argument Rule Specification

Command argument rules are used to check against certain argument values in particular commands. Argument values can be provided as a range of numerics, or a list of valid options. If a numeric range is provided, an optional exclusion range can also be provided in order to support two disjointed valid ranges. Ranges which are more complicated require a custom Python rule to support them. If FRESH finds an instance of the command stem with the specified argument either outside of the valid range, or not matching any of the options in the list, it will produce an alert of the given level.

Command argument rules use the same first four columns as other CSV flight rules, and have `Message` as the last column.

Name	Description	Format	Restrictions
FR_ID	The identifier of the flight rule.	Group-Criticality-Number	Criticality must be one of A, B, C
FR_Version	The version of the flight rule	A string of any length.	None
Alert_Level	The level of alert (violation) if the flight rule check is found to be noncompliant.	A valid FRESH FRState.	Must be one of VIOLATED, FLAGGED.
Command_Stem	The command stem of the command to be checked by the flight rule.	A valid command stem.	None
Arg_Name	The argument to be checked by the flight rule.	The name of an argument required by the command specified in command_stem.	None
Arg_Check_Type	The type of argument check to be performed by the flight rule.	A valid argument check type.	Must be one of range_int, range_float, range_hex, list_type.
Arg_Allowed_Value_List	If the flight rule check is of list_type type, the complete set of valid values for the argument.	[Value 1\ Value 2\ ...\ Value N]	Must be in a list format bracketed by square brackets and with vertical line separators.
Arg_Range	If the flight rule check is of range_int, range_float, or range_value type, the valid range of the argument.	min [<\ <=] n [<\ <=] max	Must be a range with exactly two end points, with a literal character n denoting the value of the argument in the logical statement. The numeric types should match the Arg_Check_Type for the rule.

Name	Description	Format	Restrictions
Arg_Exclusion_Range	If the flight rule check is of <code>range_int</code> , <code>range_float</code> , or <code>range_value</code> type, an optional range contained fully within the <code>Arg_Range</code> which is a set of <i>invalid</i> values. If left blank, the entire <code>Arg_Range</code> is assumed to be valid values.	<code>min [< <=] n</code> <code>[< <=] max</code>	Must be a range with exactly two end points, with a literal character <code>n</code> denoting the value of the argument in the logical statement. The numeric types should match the <code>Arg_Check_Type</code> for the rule.
Message	A message to be included when the flight rule is triggered and found to be noncompliant.	A string of any length	None

When defining the argument range, the type used for the `min` and `max` values should match the type indicated by the `Arg_Check_Type` field. Rules of type `range_int` should use integers, `range_float` should use floating point numbers (integers will be interpreted as floating point numbers), and `range_hex` should use hexadecimal numbers with a `0x` prefix. Some examples are below:

Arg_Check_Type	Examples
<code>list_type</code>	<code>[ON]</code> , <code>[STANDBY\SAFE]</code> , <code>[1\2\5]</code>
<code>range_int</code>	<code>0 < n < 6</code> , <code>-1<=n<=1</code> , <code>99 < n <= 170</code>
<code>range_float</code>	<code>1.1 <= n < 2.2</code> , <code>3 < n < 8.2</code>
<code>range_hex</code>	<code>0xAB < n < 0xCC</code>

Command Timing Rule Specification

Command timing rules are the most complex CSV rules currently available, and should only be used by developers who understand their limitations. Because command timing rules, by definition, perform checks on pairs of commands, they are subject to false positive reporting by FRESH when commands are split between sequences. FRESH only performs checks on a single sequence at a time, and cannot run its own cross-check over multiple sequences. As such, users must be cautious when interpreting command timing rule results, and developers should only use them if they understand these risks.

Command timing rules use the same first four columns as other CSV flight rules, and have `Message` as the last column.

Name	Description	Format	Restrictions
FR_ID	The identifier of the flight rule.	Group-Criticality-Number	Criticality must be one of A, B, C
FR_Version	The version of the flight rule	A string of any length.	None
Alert_Level	The level of alert (violation) if the flight rule check is found to be noncompliant.	A valid FRESH FRState.	Must be one of VIOLATED, FLAGGED.
Command_Stem	The command stem of the primary command to be checked by the flight rule.	A valid command stem.	None
Arg_Names	The argument names that must be checked to see if the the flight rule applies to an instance of the command.	"arg1;arg2;...;argN".	None

Name	Description	Format	Restrictions
Arg_Values	The values of the arguments listed in Arg_Names.	"value1;value2;...;valueN"	The list must be the same number and order as Arg_Names
Arg_Check_Type	The type of timing check to be performed for the flight rule.	A valid check type.	Must be one of follows, followed_by, waits, overlaps.
Duration_Min	The minimum number of seconds to wait before the second command. Empty means no minimum restriction.	A number of seconds.	None
Duration_Max	The maximum number of seconds to wait before the second command. Empty means no maximum restriction.	A number of seconds.	None
Rel_Command	The related command to check for. For followed_by, the command that should occur first in the sequence. For follows, waits, and overlaps, the command that should occur later in the sequence.	A valid command stem.	None
Rel_Arg_Names	The argument names that must be checked on the related command (Rel_Command) for it to meet the criteria of the flight rule.	"arg1;arg2;...;argN".	None
Rel_Arg_Values	The values of the arguments listed in Rel_Arg_Names.	"value1;value2;...;valueN"	The list must be the same number and order as Rel_Arg_Names.
Message	A message to be included when the flight rule is triggered and found to be noncompliant.	A string of any length	None

There are four types of timing rules that can be specified by the user. Each type is intended to capture the relationship between the primary command and related command in the sequence.

- **Waits:** the primary command must wait to be called no sooner than duration_min seconds after the related command occurs.
- **Overlaps:** the related command cannot execute within duration_min seconds of the primary command.
- **Follows:** the main command must occur within duration_max seconds of the related command (but still needs to wait duration_min seconds first). Such rules may also be violated if the main command follows after two or more such related commands, but is found to be too close to the later ones (violating the duration_min).
- **Followed_by:** the main command must be followed by the related command within duration_max seconds but after a duration_min wait. If no such command occurs before the sequence ends, a violation is assumed. Such rules may also be violated if the related command follows too closely to the main command, violating the duration_min.

The reason for splitting Follows and Followed_by into two rules is to allow flexibility for defining which command is reliant on the other. For Follows rules, the relevant command may occur without needing to be followed by the main command, but the main command cannot be issued without the relevant command happening first. For Followed_by rules, the relevant occur may occur without following the main command, but the main command must be followed by the relevant command after.

The arguments for timing rules can be specified in a few different ways:

1. A single literal value, checked as a string – so 4.0 may be different from 4.000.
2. A list of literal values, formatted as [arg1_value1|arg1_value2|arg1_value3] where each value is separated by a | character and the full set of values is closed with square brackets.
3. “Matching” values which compare to the other command involved in the rule using the special character =. = can be used with no other characters to match arguments with the same name (and not necessarily the same position) between the two commands. To match an argument with a different name in the related command, use =rel_arg_name with the name of the argument in the related command.

These types can be listed for multiple argument checks with a semi-colon denoting the break between each argument, for example: literal_for_arg1;[arg2_value1|arg2_value2];=;=some_arg. If multiple arguments are checked, every argument must meet the conditions specified for the rule evaluation to trigger. If any one argument specification is not met for a command, the rule will not be evaluated on that command.

If a sequence has mixed time tags, FRESH will do simple evaluation of the time using the time values of the tags. For absolute and command relative times, the command is expected to execute at the time given. Command complete times are expected to execute one second after the previous command, since no command modeling is present in FRESH. Epoch times are not evaluated. The execution length of each command is assumed to be one second.

Creating a new CSV rule type

While FRESH comes with some types of CSV flight rules ready to use, advanced users may want to create their own type of CSV rules. Creating a new type of CSV flight rules is very similar to the process for writing Python flight rules, described in the next section. CSV rules are read in by a Python class which extends the same FRBase class as other Python rules. The class should have a `check_fr` method which iterates over relevant portions of the config to read the CSV rule files. The Python class should then read each file, interpreting each rule, and performing the relevant checks on the sequence file. When all checks are complete, the `check_fr` method should return a list of FRCheckInfo objects, with each corresponding to a rule defined in the CSV.

It is recommended that any new CSV type rules keep the same first four columns, and have `Message` as the last column. This keeps information that is needed for most if not all flight rule types in the same order and format for consistency across different flight rule types. The specification for the new flight rule would then look like the below table, with additional columns added before `Message` as is needed.

Name	Description	Format	Restrictions
FR_ID	The identifier of the flight rule.	Group-Criticality-Number	Criticality must be one of A, B, C. Number is suggested to be a 0-padded four digit number.
FR_Version	The version of the flight rule	A string of any length.	None
Alert_Level	The level of alert (violation) if the flight rule check is found to be noncompliant.	A valid FRESH FRState.	Must be one of VIOLATED, FLAGGED.
Command_Stem	The command stem of the command to be checked by the flight rule.	A valid command stem.	None
Type specific columns	Any additional columns needed by the new flight rule type.
...
Message	A message to be included when the flight rule is triggered and found to be noncompliant.	A string of any length.	None

Python Rules

FRESH provides the framework for defining flight rules using Python classes. These rules cannot be configured at runtime of the tool, but they provide greater flexibility than the defined CSV rule types.

Creating a new Python rule

Creating a new Python flight rule definition is designed to be straightforward for the adapter, with no need to change any core functionality of FRESH. The steps are as follows:

1. Create a `.py` file inside the `fresh/flightrules/[MISSION IDENTIFIER]` directory.
2. Create a class that inherits from `FRBase` and implements the `check_fr` method.
3. Write any code necessary to evaluate the rule, using `check_fr` as the entry and exit point.
4. From the `check_fr` method, return a list of `FRCheckInfo` objects.

New Python flight rules should be created in a `.py` file inside the [fresh/flightrules](#) folder for the mission, where it will be automatically picked up by FRESH at runtime. Inside the file, there should be a class definition which inherits from the [FRBase](#) class and implements the `check_fr` method. Other classes can be present in the file for help with evaluation, but it is strongly recommended that only one class that inherits from `FRBase` is present per file. This class and the `check_fr` method specifically will be the entry point for evaluating the flight rule.

Adapters can make use of anything present in the `utils` folder, which includes methods and classes which may be useful for interpreting sequences or evaluating flight rules. Adapters may also write as many helper methods or classes in the flight rule file as they please, or even add new utilities to `utils` which future adapters may find useful. FRESH will only run the `check_fr` method in a class which inherits from `FRBase`, so the evaluation should start and return from this method.

If a Python flight rule requires additional information beyond the sequence to perform its checks, the FRESH pattern is to include that information in the JSON config file. Additional keys can be added to the config file to provide direct information, such as modes, or pointers to information, such as file paths, which can be read by the flight rule. The `check_fr` method provides access to a dictionary of the information in the config file plus the absolute location of the config file for resolving relative file paths. It is recommended that any prerequisite variable or other changeable information is handled in this way, so a new FRESH package is not needed every time a change is made. Constant or near-constant information included in the definition of a flight rule, such as command definitions or ranges, should be included directly in the Python file, rather than going through the config file, to reduce the burden on operators creating the config.

The `check_fr` method should return a list of `FRCheckInfo` objects, one per flight rule checked by the class. Most Python rules should return a list containing only a single `FRCheckInfo`, but flight rule classes that act as handlers for a type of flight rule definition, rather than a check on a single flight rule, may return longer lists. The returned `FRCheckInfo` object or objects will be included in the output of FRESH automatically.

FRESH core functions handle importing Python flight rule files which are present in the correct location, running `check_fr` methods with the method signature for `check_fr` defined in `FRBase`, and adding the resulting `FRCheckInfo` objects to the FRESH report.

Updating or deleting a Python rule

Python flight rules cannot be configured at runtime using the config the way the CSV rules are. The only way to update or delete a Python flight rule is by editing the source code. From the mission perspective, this means that updates to Python rules or removing a Python rule will require a new delivery of FRESH.

If an adapter or maintainer desires to remove a Python flight rule without deleting the code entirely, a leading underscore can be added to the class name. This will turn off evaluation of the rule until the leading

underscore is removed. This still requires an updated FRESH delivery to a mission, but it provides a simple way to toggle checking Python rules on or off without moving files.

Testing

Unit Testing with Pytest

The FRESH repo is set up to use pytest for unit testing all modules of the code. Custom marks are used to assign test methods to a test suite, either the core FRESH suite or a suite for an individual mission adaptation. The mark for a mission test should match the identifier used for the mission for the mission configuration. To run the tests in only a single suite, use the command `pytest -m suite`, or for multiple suites use `pytest -m "suite1 or suite2"`, with quotation marks included. For detailed information on pytest, please see the [pytest documentation](#).

The recommended pattern for creating new flight rule tests is to have one folder for each flight rule under a mission identifier folder. Within each flight rule, one or multiple test files can be present, along with one or multiple folders. Outputs that you do not wish to be committed to the repo should be placed in a folder called `out` within the flight rule folder; everything in an `out` folder will be ignored by git version tracking. Flight rule tests are allowed to assume that calls to core functionality have been independently tested and do not need to be comprehensively tested for each flight rule. However, the flight rule should test the comprehensive set of its own expected inputs and outputs to ensure the rule is performing as expected.

If a new multimission or core feature is added to FRESH, a new test suite should be added under `test/core`. Core tests should comprehensively test all parts of a new capability, and may use mission specific data structures as part of the sample data in order to do so.

Test Utils

A number of testing utilities are included in the FRESH package for ease of writing tests. They are available in [fresh/test/test_utils](#) for any test to import and use.

CLIArgs class

This class is designed to mock the CLI interface when running tests through pytest. It contains a number of methods for running relatively simple tests, and can be used by advanced users to mock the command line arguments in complicated tests.

CLIArgs.default_for_test method

Builds a default CliArgs object for a test case with one config file and one sequence file. The expected, portable way to call this method is `CliArgs.default_for_test(__file__)`, which creates an initialized CLIArgs object by looking in the same directory as `__file__` for a single file ending in the mission configured sequence file extension, and a single file ending in `.json` which is expected to be the config file.

CLIArgs.custom_for_test method

Builds a CliArgs object for a test case with the default config file, and given seq json and output files. The expected, portable way to call this method is `CliArgs.default_for_test(__file__, sequence_file, report_file)`. This method is similar to `CLIArgs.default_for_test` above, but does not look for a config file in the directory. Instead, it creates an output config file based on the default config for the mission.

CLIArgs.run_results_only_test method

Runs a simple test (see `CLIArgs.run_simple_test`), and checks that the results for flight rule `fr_id` match the step ID and state pairs given by the `steps_by_state` dictionary (see `CLIArgs.match_steps_by_state`). This can be used as an all-in-one test method when only the result state for each step needs to be checked, or as the first step in a more comprehensive test.

Returns a tuple of two values:

1. The result of `match_steps_by_state`, a dictionary with the same structure as `steps_by_state`, where step numbers are replaced by the matching result objects.
2. The `SeqDict`.

`CLIArgs.run_simple_test` method

Runs the test described by `CLIArgs.default_for_test(test_file)`, and returns a tuple of the report generated by that run and the `SeqDict` object. The expected, portable way to call this method is `run_simple_test(__file__)` from a test file in the same directory as your test inputs. It returns a dictionary of the report and the `SeqDict` object created from the input sequence. It is suggested that most flight rule tests use this method to get the results from FRESH, and subsequent checks are performed on the returned results.

`CLIArgs.run_test` method

Runs the test described by a `CLIArgs` object, and returns a tuple of the report generated by that run and the `SeqDict` object created from the input sequence.

`CLIArgs.get_step` method

Given a single result, the json object in a report corresponding with a `FRResult` object, look up the corresponding step in the `SeqDict` object.

`CLIArgs.get_arg` method

Given a list of arguments, look for an argument with a given name. If no argument with that name is found, a `KeyError` is raised.

`CLIArgs.match_to_steps` method

Attempts to match a list of results, json objects in a report corresponding with `FRResult` objects, to step numbers. This method attempts to match exactly one result object to each step number. Duplicate step numbers will be matched to different results. Fails if there is not a result for each step number, or if there is not a step number for one of the results. This method is primarily used by `CLIArgs.assert_all_steps_have_state`, but can be used independently by advanced users to create custom checks against states.

If `step_numbers` is an integer `n`, this is equivalent to `[1, 2, ..., n]`.

Returns the matching results, in the same order as the given step numbers.

`CLIArgs.assert_all_steps_have_state` method

Assert for a given flight rule that all steps in a sequence have a single result, and that result has the given state. If `steps_or_seq_dict` is a list of integers, this method will match results for those step numbers. If `steps_or_seq_dict` is a `SeqDict`, this method will match results for every step in that `seq.json`.

There is no expected return from this method. Instead, an `assert` is performed on each step to check for the given state and any failed asserts will be picked up by `pytest` as a test failure, so users of this method do not

need do to anything else to perform these checks in a pytest context.

CLIArgs.match_steps_by_state method

Attempts to match a list of results, json objects in a report corresponding with FRResult objects, to step numbers and expected states.

This method attempts to match exactly one result object to each (step number, expected state) pair. Duplicate step numbers will be matched to different results. The input argument `steps_by_state` should be a dictionary from expected state to a list of step numbers expected to have that state.

An `assert` is performed on each step to check that the state is as expected from the input and fails if there is not a result for each (step number, expected state) pair, or if there is not a pair for one of the results. Any failed asserts will be picked up by pytest as a test failure.

Returns a dictionary with the same structured as `steps_by_state`, with each step number replaced by the matching result.

Packaging and Delivery

Building the FRESH Package

FRESH is designed to be built as a python package, installable by pip. The repo comes with a `setup.py` and `setup.cfg` to be used with `setuptools` and the FRESH mission configuration. By setting the mission in [fresh/mission.py](#), the setup will pull the correct package name from the mission configuration. Different missions may have different requirements for the package name, but the FRESH installable will always create the `fresh` callable, regardless of package name.

The version of FRESH is managed using the [version.py](#) file. When creating a new package for delivery, the version number should be increased based on the type of changes performed.

Installing FRESH Locally

There are two recommended ways to install FRESH, each with their own drawbacks: pip editable mode, and a bundled distribution.

Installing in editable mode using `pip install . -e` allows for fast development because saved changes in the code will be reflected using the tool at runtime. However, using pip in this manner does not install the data files, which include CSV flight rules, default configuration files, and anything else that does not end in `.py`. Installing in this way is recommended for local development using pytest.

Installing from a bundled distribution using `pip install [package location].tar.gz` does include all the data files which are packaged with FRESH. The downside is that the package will need to be rebuilt and reinstalled for any local code changes to take effect. Installing in this way is recommended for all final checkouts before delivery, as well as when adding any new data files or changing the structure of the FRESH repo.

How to use the `build_install_test` script

A bash script is included in the FRESH repo for convenience when building and testing the package. The script takes three ordered command line arguments, and uses those arguments to run five commands. The script is intended to be run locally within your development virtual environment, and from the root of the FRESH directory. Detailed specifications are as follows:

Command line argument \$1: The package name, which should match the package name for the selected mission in [fresh/mission_config.py](#).

Command line argument \$2: The version, expected to match the value in [fresh/version.py](#) and in the same format. This is used to find the package to install.

Command line argument \$3: The mission identifier, used to run the correct suite of tests.

`pip uninstall $1`: This command uninstalls the current version of FRESH, to ensure a clean working environment.

`rm -rf build dist *.egg-info __pycache__`: This command clears old build artifacts from the root, build, and dist folders. This will remove any previously built FRESH packages.

`python setup.py sdist bdist_wheel` This command builds the FRESH package using setup.py. See the [setuptools documentation](#) for detailed information on using setuptools to build python packages. The built package will be placed in the /dist folder.

`pip install dist/$1-$2.tar.gz`: This command installs the FRESH package that was just built.

`pytest -m "core or $3" -vv`: This command runs two pytest suites, the core suite and the suite for the mission specified in the third command line argument.