ReFrame Documentation

Release 2.9

CSCS

Mar 07, 2019

Table of Contents:

ReFrame is a new framework for writing regression tests for HPC systems. The goal of this framework is to abstract away the complexity of the interactions with the system, separating the logic of a regression test from the low-level details, which pertain to the system configuration and setup. This allows users to write easily portable regression tests, focusing only on the functionality.

Regression tests in ReFrame are simple Python classes that specify the basic parameters of the test. The framework will load the test and will send it down a well-defined pipeline that will take care of its execution. The stages of this pipeline take care of all the system interaction details, such as programming environment switching, compilation, job submission, job status query, sanity checking and performance assessment.

ReFrame also offers a high-level and flexible abstraction for writing sanity and performance checks for your regression tests, without having to care about the details of parsing output files, searching for patterns and testing against reference values for different systems.

Writing system regression tests in a high-level modern programming language, like Python, poses a great advantage in organizing and maintaining the tests. Users can create their own test hierarchies or test factories for generating multiple tests at the same time and they can also customize them in a simple and expressive way.

For versions 2.6.1 and older, please refer to this documentation.

Use Cases

The ReFrame framework has been in production at CSCS since the upgrade of the Piz Daint system in early December 2016.

Read the full story...

Latest Release

Reframe is being actively developed at CSCS. You can always find the latest release here.

Publications

- *ReFrame: A regression framework for checking the health of large HPC systems* [slides]

3.1 Getting Started

3.1.1 Requirements

- Python 3.5 or higher. Python 2 is not supported.

Note: Changed in version 2.8: A functional TCL modules system is no more required. ReFrame can now operate without a modules system at all.

Optional

- For running the unit tests of the framework, the nose Python module is needed.

You are advised to run the *unit tests* of the framework after installing it on a new system to make sure that everything works fine.

3.1.2 Getting the Framework

To get the latest stable version of the framework, you can just clone it from the github project page:

```
git clone https://github.com/eth-cscs/reframe.git
```

Alternatively, you can pick a previous stable version by downloading it from the previous releases section.

3.1.3 Running the Unit Tests

After you have downloaded the framework, it is important to run the unit tests of to make sure that everything is set up correctly:

```
./test_reframe.py -v
```

The output should look like the following:

```
test_check_failure (unittests.test_cli.TestFrontend) ... ok
test_check_sanity_failure (unittests.test_cli.TestFrontend) ... ok
test_check_submit_success (unittests.test_cli.TestFrontend) ... SKIP: job submission␣
↪not supported
test_check_success (unittests.test_cli.TestFrontend) ... ok
test_checkpath_recursion (unittests.test_cli.TestFrontend) ... ok
test_custom_performance_check_failure (unittests.test_cli.TestFrontend) ... ok
...
test_copytree (unittests.test_utility.TestOSTools) ... ok
test_grep (unittests.test_utility.TestOSTools) ... ok
test_inpath (unittests.test_utility.TestOSTools) ... ok
test_subdirs (unittests.test_utility.TestOSTools) ... ok
test_always_true (unittests.test_utility.TestUtilityFunctions) ... ok
test_standard_threshold (unittests.test_utility.TestUtilityFunctions) ... ok


----------------------------------------------------------------------
Ran 235 tests in 33.842s

OK (SKIP=7)
```

You will notice in the output that all the job submission related tests have been skipped. The test suite detects if the current system has a job submission system and is configured for ReFrame (see Configuring ReFrame for your site) and it will skip all the unsupported unit tests. As soon as you configure ReFrame for your system, you can rerun the test suite to check that job submission unit tests pass as well. Note here that some unit tests may still be skipped depending on the configured job submission system.

3.1.4 Where to Go from Here

The next step from here is to setup and configure ReFrame for your site, so that ReFrame can automatically recognize it and submit jobs. Please refer to the "Configuring ReFrame For Your Site" section on how to do that.

Before starting implementing a regression test, you should go through the "The Regression Test Pipeline" section, so as to understand the mechanism that ReFrame uses to run the regression tests. This section will let you follow easily the "ReFrame Tutorial" as well as understand the more advanced examples in the "Customizing Further A Regression Test" section.

To learn how to invoke the ReFrame command-line interface for running your tests, please refer to the "Running ReFrame" section.

3.2 Configuring ReFrame for Your Site

ReFrame provides an easy and flexible way to configure new systems and new programming environments. By default, it ships with a generic local system configured. This should be enough to let you run ReFrame on a local computer as soon as the basic software requirements are met.

As soon as a new system with its programming environments is configured, adapting an existing regression test could be as easy as just adding the system's name in the *valid_systems* list and its associated programming environments in the *valid_prog_environs* list.

3.2.1 The Configuration File

The configuration of systems and programming environments is performed by a special Python dictionary called _site_configuration defined inside the file <install-dir>/reframe/settings.py.

The _site_configuration dictionary should define two entries, systems and environments. The former defines the systems that ReFrame may recognize, whereas the latter defines the available programming environments.

The following example shows a minimal configuration for the Piz Daint supercomputer at CSCS:

```python
_site_configuration = {
    'systems': {
        'daint': {
            'descr': 'Piz Daint',
            'hostnames': ['daint'],
            'modules_system': 'tmod',
            'partitions': {
                'login': {
                    'scheduler': 'local',
                    'modules': [],
                    'access':  [],
                    'environs': ['PrgEnv-cray', 'PrgEnv-gnu',
                                 'PrgEnv-intel', 'PrgEnv-pgi'],
                    'descr': 'Login nodes',
                    'max_jobs': 4
                },

                'gpu': {
                    'scheduler': 'nativeslurm',
                    'modules': ['daint-gpu'],
                    'access':  ['--constraint=gpu'],
                    'environs': ['PrgEnv-cray', 'PrgEnv-gnu',
                                 'PrgEnv-intel', 'PrgEnv-pgi'],
                    'descr': 'Hybrid nodes (Haswell/P100)',
                    'max_jobs': 100
                },

                'mc': {
                    'scheduler': 'nativeslurm',
                    'modules': ['daint-mc'],
                    'access':  ['--constraint=mc'],
                    'environs': ['PrgEnv-cray', 'PrgEnv-gnu',
                                 'PrgEnv-intel', 'PrgEnv-pgi'],
                    'descr': 'Multicore nodes (Broadwell)',
                    'max_jobs': 100
                }
            }
        }
    },

    'environments': {
        '*': {
            'PrgEnv-cray': {
```

```
            'type': 'ProgEnvironment',
            'modules': ['PrgEnv-cray'],
        },

        'PrgEnv-gnu': {
            'type': 'ProgEnvironment',
            'modules': ['PrgEnv-gnu'],
        },

        'PrgEnv-intel': {
            'type': 'ProgEnvironment',
            'modules': ['PrgEnv-intel'],
        },

        'PrgEnv-pgi': {
            'type': 'ProgEnvironment',
            'modules': ['PrgEnv-pgi'],
        }
    }
}
}
```

3.2.2 System Configuration

The list of supported systems is defined as a set of key/value pairs under key `systems`. Each system is a key/value pair, with the key being the name of the system and the value being another set of key/value pairs defining its attributes. The valid attributes of a system are the following:

- `descr`: A detailed description of the system (default is the system name).

- `hostnames`: This is a list of hostname patterns that will be used by ReFrame when it tries to *auto-detect* the current system (default `[]`).

- `modules_system`: The modules system that should be used for loading environment modules on this system. The only available modules system backend is currently `tmod`, which corresponds to the TCL implementation of the environment modules (default `None`).

- `prefix`: Default regression prefix for this system (default `.`).

- `stagedir`: Default stage directory for this system (default `None`).

- `outputdir`: Default output directory for this system (default `None`).

- `logdir`: Default performance logging directory for this system (default `None`).

- `resourcesdir`: Default directory for storing large resources (e.g., input data files, etc.) needed by regression tests for this system (default `.`).

- `partitions`: A set of key/value pairs defining the partitions of this system and their properties (default `{}`). Partition configuration is discussed in the *next section*.

Note: New in version 2.8: The `modules_system` key was introduced for specifying custom modules systems for different systems.

For a more detailed description of the `prefix`, `stagedir`, `outputdir` and `logdir` directories, please refer to the "Running ReFrame" section.

3.2.3 Partition Configuration

From the ReFrame's point of view, each system consists of a set of logical partitions. These partitions need not necessarily correspond to real scheduler partitions. For example, Piz Daint on the above example is split in *virtual partitions* using Slurm constraints. Other systems may be indeed split into real scheduler partitions.

The partitions of a system are defined similarly to systems as a set of key/value pairs with the key being the partition name and the value being another set of key/value pairs defining the partition's attributes. The available partition attributes are the following:

- `descr`: A detailed description of the partition (default is the partition name).

- `scheduler`: The job scheduler and parallel program launcher combination that is used on this partition to launch jobs. The syntax of this attribute is `<scheduler>+<launcher>`. The available values for the job scheduler are the following:

    - `slurm`: Jobs on this partition will be launched using Slurm. This scheduler relies on job accounting (`sacct` command) in order to reliably query the job status.

    - `squeue`: Jobs on this partition will be launched using Slurm, but no job accounting is required. The job status is obtained using the `squeue` command. This scheduler is less reliable than the one based on the `sacct` command, but the framework does its best to query the job state as reliably as possible.

    - `local`: Jobs on this partition will be launched locally as OS processes.

    The available values for the parallel program launchers are the following:

    - `srun`: Programs on this partition will be launched using a bare `srun` command *without* any job allocation options passed to it. This launcher may only be used with the `slurm` scheduler.

    - `srunalloc`: Programs on this partition will be launched using the `srun` command *with* job allocation options passed automatically to it. This launcher may also be used with the `local` scheduler.

    - `alps`: Programs on this partition will be launched using the `aprun` command.

    - `mpirun`: Programs on this partition will be launched using the `mpirun` command.

    - `mpiexec`: Programs on this partition will be launched using the `mpiexec` command.

    - `local`: Programs on this partition will be launched as-is without using any parallel program launcher.

    There exist also the following aliases for specific combinations of job schedulers and parallel program launchers:

    - `nativeslurm`: This is equivalent to `slurm+srun`.

    - `local`: This is equivalent to `local+local`.

- `access`: A list of scheduler options that will be passed to the generated job script for gaining access to that logical partition (default `[]`).

- `environs`: A list of environments, with which ReFrame will try to run any regression tests written for this partition (default `[]`). The environment names must be resolved inside the `environments` section of the `_site_configuration` dictionary (see *Environments Configuration* for more information).

- `modules`: A list of modules to be loaded before running a regression test on that partition (default `[]`).

- `variables`: A set of environment variables to be set before running a regression test on that partition (default `{}`). Environment variables can be set as follows (notice that both the variable name and its value are strings):

```
'variables': {
    'MYVAR': '3',
    'OTHER': 'foo'
}
```

- `max_jobs`: The maximum number of concurrent regression tests that may be active (not completed) on this partition. This option is relevant only when ReFrame executes with the asynchronous execution policy.

- `resources`: A set of custom resource specifications and how these can be requested from the partition's scheduler (default `{}`).

  This variable is a set of key/value pairs with the key being the resource name and the value being a list of options to be passed to the partition's job scheduler. The option strings can contain *placeholders* of the form `{placeholder_name}`. These placeholders may be replaced with concrete values by a regression tests through the `extra_resources` attribute.

  For example, one could define a `gpu` resource for a multi-GPU system that uses Slurm as follows:

```
'resources': {
    'gpu': ['--gres=gpu:{num_gpus_per_node}']
}
```

  A regression test then may request this resource as follows:

```
self.extra_resources = {'gpu': {'num_gpus_per_node': '8'}}
```

  And the generated job script will have the following line in its preamble:

```
#SBATCH --gres=gpu:8
```

  A resource specification may also start with `#PREFIX`, in which case `#PREFIX` will replace the standard job script prefix of the backend scheduler of this partition. This is useful in cases of job schedulers like Slurm, that allow alternative prefixes for certain features. An example is the DataWarp functionality of Slurm which is supported by the `#DW` prefix. One could then define DataWarp related resources as follows:

```
'resources': {
    'datawarp': [
        '#DW jobdw capacity={capacity} access_mode={mode} type=scratch'
        '#DW stage_out source={out_src} destination={out_dst} type={stage_
↪filetype}'
    ]
}
```

  A regression test that wants to make use of that resource, it can set its `extra_resources` as follows:

```
self.extra_resources = {
    'datawarp': {
        'capacity': '100GB',
        'mode': 'striped',
        'out_src': '$DW_JOB_STRIPED/name',
        'out_dst': '/my/file',
        'stage_filetype': 'file'
    }
}
```

Note: Changed in version 2.8: A new syntax for the `scheduler` values was introduced as well as more parallel program launchers. The old values for the `scheduler` key will continue to be supported.

Note: New in version 2.8.1: The `squeue` backend scheduler was added.

Note: Changed in version 2.9: Better support for custom job resources.

3.2.4 Environments Configuration

The environments available for testing in different systems are defined under the `environments` key of the top-level `_site_configuration` dictionary. The `environments` key is associated to a special dictionary that defines scopes for looking up an environment. The `*` denotes the global scope and all environments defined there can be used by any system. Instead of `*`, you can define scopes for specific systems or specific partitions by using the name of the system or partition. For example, an entry `daint` will define a scope for a system called `daint`, whereas an entry `daint:gpu` will define a scope for a virtual partition named `gpu` on the system `daint`. When an environment name is used in the `environs` list of a system partition (see *Partition Configuration*), it is first looked up in the entry of that partition, e.g., `daint:gpu`. If no such entry exists, it is looked up in the entry of the system, e.g., `daint`. If not found there, it is looked up in the global scope denoted by the `*` key. If it cannot be found even there, an error will be issued. This look up mechanism allows you to redefine an environment for a specific system or partition. In the following example, we redefine `PrgEnv-gnu` for a system named `foo`, so that whenever `PrgEnv-gnu` is used on that system, the module `openmpi` will also be loaded and the compiler variables should point to the MPI wrappers.

```
'foo': {
    'PrgEnv-gnu': {
        'type': 'ProgEnvironment',
        'modules': ['PrgEnv-gnu', 'openmpi'],
        'cc':  'mpicc',
        'cxx': 'mpicxx',
        'ftn': 'mpif90',
    }
}
```

An environment is also defined as a set of key/value pairs with the key being its name and the value being a dictionary of its attributes. The possible attributes of an environment are the following:

- `type`: The type of the environment to create. There are two available environment types (note that names are case sensitive):

    - `'Environment'`: A simple environment.

    - `'ProgEnvironment'`: A programming environment.

- `modules`: A list of modules to be loaded when this environment is used (default `[]`, valid for all environment types)

- `variables`: A set of variables to be set when this environment is used (default `{}`, valid for all environment types)

- `cc`: The C compiler (default `'cc'`, valid for `'ProgEnvironment'` only).

- `cxx`: The C++ compiler (default `'CC'`, valid for `'ProgEnvironment'` only).

- `ftn`: The Fortran compiler (default `'ftn'`, valid for `'ProgEnvironment'` only).

- `cppflags`: The default preprocessor flags (default `None`, valid for `'ProgEnvironment'` only).

- `cflags`: The default C compiler flags (default `None`, valid for `'ProgEnvironment'` only).

- `cxxflags`: The default C++ compiler flags (default `None`, valid for `'ProgEnvironment'` only).

- `fflags`: The default Fortran compiler flags (default `None`, valid for `'ProgEnvironment'` only).

- `ldflags`: The default linker flags (default `None`, valid for `'ProgEnvironment'` only).

Note: When defining programming environment flags, `None` is treated differently from `''` for regression tests that are compiled through a Makefile. If a flags variable is not `None` it will be passed to the Makefile, which may affect the compilation process.

3.2.5 System Auto-Detection

When the ReFrame is launched, it tries to auto-detect the current system based on its site configuration. The auto-detection process is as follows:

ReFrame first tries to obtain the hostname from `/etc/xthostname`, which provides the unqualified *machine name* in Cray systems. If this cannot be found the hostname will be obtained from the standard `hostname` command. Having retrieved the hostname, ReFrame goes through all the systems in its configuration and tries to match the hostname against any of the patterns in the `hostnames` attribute of *system configuration*. The detection process stops at the first match found, and the system it belongs to is considered as the current system. If the system cannot be auto-detected, ReFrame will fail with an error message. You can override completely the auto-detection process by specifying a system or a system partition with the `--system` option (e.g., `--system daint` or `--system daint:gpu`).

3.3 The Regression Test Pipeline

The backbone of the ReFrame regression framework is the regression test pipeline. This is a set of well defined phases that each regression test goes through during its lifetime. The figure below depicts this pipeline in detail.

Fig. 1: The regression test pipeline

A regression test starts its life after it has been instantiated by the framework. This is where all the basic information of the test is set. At this point, although it is initialized, the regression test is not yet *live*, meaning that it does not run yet. The framework will then go over all the loaded and initialized checks (we will talk about the loading and selection phases later), it will pick the next partition of the current system and the next programming environment for testing and will try to run the test. If the test supports the current system partition and the current programming environment, it will be run and it will go through all the following seven phases:

1. Setup
2. Compilation
3. Running
4. Sanity checking
5. Performance checking
6. Cleanup

A test may implement some of them as no-ops. As soon as the test is finished, its resources are cleaned up and the framework's environment is restored. ReFrame will try to repeat the same procedure on the same regression test using the next programming environment and the next system partition until no further environments and partitions are left to be tested. In the following we elaborate on each of the individual phases of the lifetime of a regression test.

3.3.1 0. The Initialization Phase

This phase is not part of the regression test pipeline as shown above, but it is quite important, since during this phase the test is loaded into memory and initialized. As we shall see in the "Tutorial" and in the "Customizing Further A ReFrame Regression Test" sections, this is the phase where the *specification* of a test is set. At this point the current system is already known and the test may be set up accordingly. If no further differentiation is needed depending on the system partition or the programming environment, the test could go through the whole pipeline performing all of its work without the need to override any of the other pipeline stages. In fact, this is perhaps the most common case for most of the regression tests.

3.3.2 1. The Setup Phase

A regression test is instantiated once by the framework and it is then copied each time a new system partition or programming environment is tried. This first phase of the regression pipeline serves the purpose of preparing the test to run on the specified partition and programming environment by performing a number of operations described below:

Set up and load the test's environment

At this point the environment of the current partition, the current programming environment and any test's specific environment will be loaded. For example, if the current partition requires `slurm`, the current programming environment is `PrgEnv-gnu` and the test requires also `cudatoolkit`, this phase will be equivalent to the following:

```
module load slurm
module unload PrgEnv-cray
module load PrgEnv-gnu
module load cudatoolkit
```

Note that the framework automatically detects conflicting modules and unloads them first. So the user need not to care about the existing environment at all. She only needs to specify what is needed by her test.

Setup the test's paths

Each regression test is associated with a stage directory and an output directory. The stage directory will be the working directory of the test and all of its resources will be copied there before running. The output directory is the directory where some important output files of the test will be kept. By default these are the generated job script file, the standard output and standard error. The user can also specify additional files to be kept in the test's specification. At this phase, all these directories are created.

Prepare a job for the test

At this point a *job descriptor* will be created for the test. A job descriptor in ReFrame is an abstraction of the job scheduler's functionality relevant to the regression framework. It is responsible for submitting a job in a job queue and waiting for its completion. ReFrame supports two job scheduler backends that can be combined with several different parallel program launchers. For a complete list of the job scheduler/parallel launchers combinations, please refer to "Partition Configuration".

3.3.3 2. The Compilation Phase

At this phase the source code associated with test is compiled with the current programming environment. Before compiling, all the resources of the test are copied to its stage directory and the compilation is performed from that

directory.

3.3.4 3. The Run Phase

This phase comprises two subphases:

- **Job launch**: At this subphase a job script file for the regression test is generated and submitted to the job scheduler queue. If the job scheduler for the current partition is the **local** one, a simple wrapper shell script will be generated and will be launched as a local OS process.

- **Job wait**: At this subphase the job (or local process) launched in the previous subphase is waited for. This phase is pretty basic: it just checks that the launched job (or local process) has finished. No check is made of whether the job or process has finished successfully or not. This is the responsibility of the next pipeline stage.

ReFrame currently supports two execution policies:

- **serial**: In the serial execution policy, these two subphases are performed back-to-back and the framework blocks until the current regression test finishes.

- **asynchronous**: In the asynchronous execution policy, as soon as the job associated to the current test is launched, ReFrame continues its execution by executing and launching the subsequent test cases.

3.3.5 4. The Sanity Checking Phase

At this phase it is determined whether the check has finished successfully or not. Although this decision is test-specific, ReFrame provides a very flexible and expressive way for specifying complex patterns and operations to be performed on the test's output in order to determine the outcome of the test.

3.3.6 5. The Performance Checking Phase

At this phase the performance of the regression test is checked. ReFrame uses the same mechanism for analyzing the output of the test as with sanity checking. The only difference is that the user can now specify reference values per system or system partition, as well as acceptable performance thresholds

3.3.7 6. The Cleanup Phase

This is the final stage of the regression test pipeline and it is responsible for cleaning up the resources of the test. Three steps are performed in this phase:

1. The interesting files of the test (job script, standard output and standard error and any additional files specified by the user) are copied to its output directory for later inspection and bookkeeping,

2. the stage directory is removed and

3. the test's environment is revoked.

At this point the ReFrame's environment is clean and in its original state and the framework may continue by running more test cases.

3.4 ReFrame Tutorial

This tutorial will guide you through writing your first regression tests with ReFrame. We will start with the most common and simple case of a regression test that compiles a code, runs it and checks its output. We will then expand

this example gradually by adding functionality and more advanced sanity and performance checks. By the end of the tutorial, you should be able to start writing your first regression tests with ReFrame.

If you just want to get a quick feeling of how it is like writing a regression test in ReFrame, you can start directly from here. However, if you want to get a better understanding of what is happening behind the scenes, we recommend to have a look also in "The Regression Test Pipeline" section.

All the tutorial examples can be found in `<reframe-install-prefix>/tutorial/`.

For the configuration of the system, we provide a minimal configuration file for Piz Daint, where we have tested all the tutorial examples. The site configuration that we used for this tutorial is the following:

```python
...
_site_configuration = {
    'systems': {
        'daint': {
            'descr': 'Piz Daint',
            'hostnames': ['daint'],
            'partitions': {
                'login': {
                    'scheduler': 'local',
                    'modules': [],
                    'access':  [],
                    'environs': ['PrgEnv-cray', 'PrgEnv-gnu',
                                 'PrgEnv-intel', 'PrgEnv-pgi'],
                    'descr': 'Login nodes',
                    'max_jobs': 4
                },

                'gpu': {
                    'scheduler': 'nativeslurm',
                    'modules': ['daint-gpu'],
                    'access':  ['--constraint=gpu'],
                    'environs': ['PrgEnv-cray', 'PrgEnv-gnu',
                                 'PrgEnv-intel', 'PrgEnv-pgi'],
                    'descr': 'Hybrid nodes (Haswell/P100)',
                    'max_jobs': 100
                },

                'mc': {
                    'scheduler': 'nativeslurm',
                    'modules': ['daint-mc'],
                    'access':  ['--constraint=mc'],
                    'environs': ['PrgEnv-cray', 'PrgEnv-gnu',
                                 'PrgEnv-intel', 'PrgEnv-pgi'],
                    'descr': 'Multicore nodes (Broadwell)',
                    'max_jobs': 100
                }
            }
        }
    },

    'environments': {
        '*': {
            'PrgEnv-cray': {
                'type': 'ProgEnvironment',
                'modules': ['PrgEnv-cray'],
            },
```

(continues on next page)

```
        'PrgEnv-gnu': {
            'type': 'ProgEnvironment',
            'modules': ['PrgEnv-gnu'],
        },

        'PrgEnv-intel': {
            'type': 'ProgEnvironment',
            'modules': ['PrgEnv-intel'],
        },

        'PrgEnv-pgi': {
            'type': 'ProgEnvironment',
            'modules': ['PrgEnv-pgi'],
        }
    }
  }
}
...
```

You can find the full `settings.py` file ready to be used by ReFrame in `<reframe-install-prefix>/tutorial/config/settings.py`. You may first need to go over the "Configuring ReFrame For Your Site" section, in order to prepare the framework for your systems.

3.4.1 The First Regression Test

The following is a simple regression test that compiles and runs a serial C program, which computes a matrix-vector product (`tutorial/src/example_matrix_multiplication.c`), and verifies its sane execution. As a sanity check, it simply looks for a specific output in the output of the program. Here is the full code for this test:

```python
import os
import reframe.utility.sanity as sn

from reframe.core.pipeline import RegressionTest


class SerialTest(RegressionTest):
    def __init__(self, **kwargs):
        super().__init__('example1_check',
                         os.path.dirname(__file__), **kwargs)
        self.descr = 'Simple matrix-vector multiplication example'
        self.valid_systems = ['*']
        self.valid_prog_environs = ['*']
        self.sourcepath = 'example_matrix_vector_multiplication.c'
        self.executable_opts = ['1024', '100']
        self.sanity_patterns = sn.assert_found(
            r'time for single matrix vector multiplication', self.stdout)
        self.maintainers = ['you-can-type-your-email-here']
        self.tags = {'tutorial'}


def _get_checks(**kwargs):
    return [SerialTest(**kwargs)]
```

A regression test written in ReFrame is essentially a Python class that must eventually derive from *RegressionTest*. In order to make the test available to the framework, every file defining regression tests must

---

define the special function `_get_checks()`, which should return a list of instantiated regression tests. This method will be called by the framework upon loading your file, in order to retrieve the regression tests defined. The framework will pass some special arguments to the `_get_checks()` function through the `kwargs` parameter, which are needed for the correct initialization of the regression test.

Now let's move on to the actual definition of the `SerialTest` here:

```python
class SerialTest(RegressionTest):
    def __init__(self, **kwargs):
        super().__init__('example1_check', os.path.dirname(__file__), **kwargs)
```

The `__init__()` method is the constructor of your test. It is usually the only method you need to implement for your tests, especially if you don't want to customize any of the regression test pipeline stages. The first statement in the `SerialTest` constructor calls the constructor of the base class, passing as arguments the name of the regression test (`example1_check` here), the path to the test directory and any other arguments passed to the `SerialTest`'s constructor. You can consider these first three lines and especially the way you should call the constructor of the base class, as boilerplate code. As you will see, it remains the same across all our examples, except, of course, for the check name.

The next line sets a more detailed description of the test:

```python
self.descr = 'Simple matrix-vector multiplication example'
```

This is optional and it defaults to the regression test's name, if not specified.

The next two lines specify the systems and the programming environments that this test is valid for:

```python
self.valid_systems = ['*']
self.valid_prog_environs = ['*']
```

Both of these variables accept a list of system names or environment names, respectively. The `*` symbol is a wildcard meaning any system or any programming environment. The system and environment names listed in these variables must correspond to names of systems and environments defined in the ReFrame's settings file.

When specifying system names you can always specify a partition name as well by appending `:<partname>` to the system's name. For example, given the configuration for our tutorial, `daint:gpu` would refer specifically to the `gpu` virtual partition of the system `daint`. If only a system name (without a partition) is specified in the *`self.valid_systems`* variable, e.g., `daint`, it means that this test is valid for any partition of this system.

The next line specifies the source file that needs to be compiled:

```python
self.sourcepath = 'example_matrix_vector_multiplication.c'
```

ReFrame expects any source files, or generally resources, of the test to be inside an `src/` directory, which is at the same level as the regression test file. If you inspect the directory structure of the `tutorial/` folder, you will notice that:

```
tutorial/
    example1.py
    src/
        example_matrix_vector_multiplication.c
```

Notice also that you need not specify the programming language of the file you are asking ReFrame to compile or the compiler to use. ReFrame will automatically pick the correct compiler based on the extension of the source file. The exact compiler that is going to be used depends on the programming environment that the test is running with. For example, given our configuration, if it is run with `PrgEnv-cray`, the Cray C compiler will be used, if it is run with `PrgEnv-gnu`, the GCC compiler will be used etc. A user can associate compilers with programming environments in the ReFrame's settings file.

The next line in our first regression test specifies a list of options to be used for running the generated executable (the matrix dimension and the number of iterations in this particular example):

```
self.executable_opts = ['1024', '100']
```

Notice that you do not need to specify the executable name. Since ReFrame compiled it and generated it, it knows the name. We will see in the "Customizing Further A ReFrame Regression Test" section, how you can specify the name of the executable, in cases that ReFrame cannot guess its name.

The next lines specify what should be checked for assessing the sanity of the result of the test:

```
self.sanity_patterns = sn.assert_found(
    r'time for single matrix vector multiplication', self.stdout)
```

This expression simply asks ReFrame to look for `time for single matrix vector multiplication` in the standard output of the test. The `sanity_patterns` attribute can only be assigned the result of a special type of functions, called *sanity functions*. Sanity functions are special in the sense that they are evaluated lazily. You can generally treat them as normal Python functions inside a `sanity_patterns` expression. ReFrame provides already a wide range of useful sanity functions ranging from wrappers to the standard built-in functions of Python to functions related to parsing the output of a regression test. For a complete listing of the available functions, please have a look at the "Sanity Functions Reference".

In our example, the `assert_found` function accepts a regular expression pattern to be searched in a file and either returns `True` on success or raises a `SanityError` in case of failure with a descriptive message. This function uses internally the "re" module of the Python standard library, so it may accept the same regular expression syntax. As a file argument, `assert_found` accepts any filename, which will be resolved against the stage directory of the test. You can also use the `stdout` and `stderr` attributes to reference the standard output and standard error, respectively.

Note: You need not to care about handling exceptions, and error handling in general, inside your test. The framework will automatically abort the execution of the test, report the error and continue with the next test case.

The last two lines of the regression test are optional, but serve a good role in a production environment:

```
self.maintainers = ['you-can-type-your-email-here']
self.tags = {'tutorial'}
```

In the `maintainers` attribute you may store a list of people responsible for the maintenance of this test. In case of failure, this list will be printed in the failure summary.

The `tags` attribute is a set of tags that you can assign to this test. This is useful for categorizing the tests and helps in quickly selecting the tests of interest. More about test selection, you can find in the "Running ReFrame" section.

Note: The values assigned to the attributes of a `RegressionTest` are validated and if they don't have the correct type, an error will be issued by ReFrame. For a list of all the attributes and their types, please refer to the "Reference Guide".

Running the Tutorial Examples

ReFrame offers a rich command-line interface that allows you to control several aspects of its executions. A more detailed description can be found in the "Running ReFrame" section. Here we will only show you how to run a specific tutorial test:

```
./bin/reframe -c tutorial/ -n example1_check -r
```

If everything is configured correctly for your system, you should get an output similar to the following:

```
Reframe version: 2.7
Launched by user: <your-username>
Launched on host: daint104
Reframe paths
=============
    Check prefix      :
    Check search path : 'tutorial/'
    Stage dir prefix  : <cwd>/stage/
    Output dir prefix : <cwd>/output/
    Logging dir       : <cwd>/logs
[==========] Running 1 check(s)
[==========] Started on Fri Oct 20 15:11:38 2017

[----------] started processing example1_check (Simple matrix-vector multiplication␣
↪example)
[ RUN      ] example1_check on daint:mc using PrgEnv-cray
[       OK ] example1_check on daint:mc using PrgEnv-cray
[ RUN      ] example1_check on daint:mc using PrgEnv-gnu
[       OK ] example1_check on daint:mc using PrgEnv-gnu
[ RUN      ] example1_check on daint:mc using PrgEnv-intel
[       OK ] example1_check on daint:mc using PrgEnv-intel
[ RUN      ] example1_check on daint:mc using PrgEnv-pgi
[       OK ] example1_check on daint:mc using PrgEnv-pgi
[ RUN      ] example1_check on daint:login using PrgEnv-cray
[       OK ] example1_check on daint:login using PrgEnv-cray
[ RUN      ] example1_check on daint:login using PrgEnv-gnu
[       OK ] example1_check on daint:login using PrgEnv-gnu
[ RUN      ] example1_check on daint:login using PrgEnv-intel
[       OK ] example1_check on daint:login using PrgEnv-intel
[ RUN      ] example1_check on daint:login using PrgEnv-pgi
[       OK ] example1_check on daint:login using PrgEnv-pgi
[ RUN      ] example1_check on daint:gpu using PrgEnv-cray
[       OK ] example1_check on daint:gpu using PrgEnv-cray
[ RUN      ] example1_check on daint:gpu using PrgEnv-gnu
[       OK ] example1_check on daint:gpu using PrgEnv-gnu
[ RUN      ] example1_check on daint:gpu using PrgEnv-intel
[       OK ] example1_check on daint:gpu using PrgEnv-intel
[ RUN      ] example1_check on daint:gpu using PrgEnv-pgi
[       OK ] example1_check on daint:gpu using PrgEnv-pgi
[----------] finished processing example1_check (Simple matrix-vector multiplication␣
↪example)

[  PASSED  ] Ran 12 test case(s) from 1 check(s) (0 failure(s))
[==========] Finished on Fri Oct 20 15:15:25 2017
```

Notice how our regression test is run on every partition of the configured system and for every programming environment.

Now that you have got a first understanding of how a regression test is written in ReFrame, let's try to expand our example.

3.4.2 Customizing the Compilation Phase

In this example, we write a regression test to compile and run the OpenMP version of the matrix-vector product program, that we have shown before. The full code of this test follows:

```python
import os
import reframe.utility.sanity as sn

from reframe.core.pipeline import RegressionTest


class OpenMPTestIfElse(RegressionTest):
    def __init__(self, **kwargs):
        super().__init__('example2a_check',
                         os.path.dirname(__file__), **kwargs)
        self.descr = 'Matrix-vector multiplication example with OpenMP'
        self.valid_systems = ['*']
        self.valid_prog_environs = ['PrgEnv-cray', 'PrgEnv-gnu',
                                    'PrgEnv-intel', 'PrgEnv-pgi']
        self.sourcepath = 'example_matrix_vector_multiplication_openmp.c'
        self.executable_opts = ['1024', '100']
        self.variables = {
            'OMP_NUM_THREADS': '4'
        }
        self.sanity_patterns = sn.assert_found(
            r'time for single matrix vector multiplication', self.stdout)
        self.maintainers = ['you-can-type-your-email-here']
        self.tags = {'tutorial'}

    def compile(self):
        env_name = self.current_environ.name
        if env_name == 'PrgEnv-cray':
            self.current_environ.cflags = '-homp'
        elif env_name == 'PrgEnv-gnu':
            self.current_environ.cflags = '-fopenmp'
        elif env_name == 'PrgEnv-intel':
            self.current_environ.cflags = '-openmp'
        elif env_name == 'PrgEnv-pgi':
            self.current_environ.cflags = '-mp'

        super().compile()


def _get_checks(**kwargs):
    return [OpenMPTestIfElse(**kwargs)]
```

This example introduces two new concepts:

1. We need to set the `OMP_NUM_THREADS` environment variable, in order to specify the number of threads to use with our program.

2. We need to specify different flags for the different compilers provided by the programming environments we are testing. Notice also that we now restrict the validity of our test only to the programming environments that we know how to handle (see the `valid_prog_environs`).

To define environment variables to be set during the execution of a test, you should use the `variables` attribute of the `RegressionTest` class. This is a dictionary, whose keys are the names of the environment variables and whose values are the values of the environment variables. Notice that both the keys and the values must be strings.

In order to set the compiler flags for the current programming environment, you have to override either the *setup* or the *compile* method of the *RegressionTest*. As described in "The Regression Test Pipeline" section, it is during the setup phase that a regression test is prepared for a new system partition and a new programming environment. Here we choose to override the compile() method, since setting compiler flags is simply more relevant to this phase conceptually.

Note: The *RegressionTest* implements the six phases of the regression test pipeline in separate methods. Individual regression tests may override them to provide alternative implementations, but in all practical cases, only the *setup* and the *compile* methods may need to be overriden. You will hardly ever need to override any of the other methods and, in fact, you should be very careful when doing it.

The *current_environ* attribute of the *RegressionTest* holds an instance of the current programming environment. This variable is available to regression tests after the setup phase. Before it is None, so you cannot access it safely during the initialization phase. Let's have a closer look at the compile() method:

```python
def compile(self):
    env_name = self.current_environ.name
    if env_name == 'PrgEnv-cray':
        self.current_environ.cflags = '-homp'
    elif env_name == 'PrgEnv-gnu':
        self.current_environ.cflags = '-fopenmp'
    elif env_name == 'PrgEnv-intel':
        self.current_environ.cflags = '-openmp'
    elif env_name == 'PrgEnv-pgi':
        self.current_environ.cflags = '-mp'

    super().compile()
```

We first take the name of the current programming environment (self.current_environ.name) and we check it against the set of the known programming environments. We then set the compilation flags accordingly. Since our target file is a C program, we just set the cflags of the current programming environment. Finally, we call the compile() method of the base class, in order to perform the actual compilation.

An alternative implementation using dictionaries

Here we present an alternative implementation of the same test using a dictionary to hold the compilation flags for the different programming environments. The advantage of this implementation is that you move the different compilation flags in the initialization phase, where also the rest of the test's specification is, thus making it more concise.

The compile() method is now very simple: it gets the correct compilation flags from the prgenv_flags dictionary and applies them to the current programming environment.

Note: A regression test is like any other Python class, so you can freely define your own attributes. If you accidentally try to write on a reserved *RegressionTest* attribute that is not writeable, ReFrame will prevent this and it will throw an error.

```python
import os
import reframe.utility.sanity as sn

from reframe.core.pipeline import RegressionTest
```

(continues on next page)

```python
class OpenMPTestDict(RegressionTest):
    def __init__(self, **kwargs):
        super().__init__('example2b_check',
                         os.path.dirname(__file__), **kwargs)
        self.descr = 'Matrix-vector multiplication example with OpenMP'
        self.valid_systems = ['*']
        self.valid_prog_environs = ['PrgEnv-cray', 'PrgEnv-gnu',
                                    'PrgEnv-intel', 'PrgEnv-pgi']
        self.sourcepath = 'example_matrix_vector_multiplication_openmp.c'
        self.executable_opts = ['1024', '100']
        self.prgenv_flags = {
            'PrgEnv-cray':  '-homp',
            'PrgEnv-gnu':   '-fopenmp',
            'PrgEnv-intel': '-openmp',
            'PrgEnv-pgi':   '-mp'
        }
        self.variables = {
            'OMP_NUM_THREADS': '4'
        }
        self.sanity_patterns = sn.assert_found(
            r'time for single matrix vector multiplication', self.stdout)
        self.maintainers = ['you-can-type-your-email-here']
        self.tags = {'tutorial'}

    def compile(self):
        prgenv_flags = self.prgenv_flags[self.current_environ.name]
        self.current_environ.cflags = prgenv_flags
        super().compile()


def _get_checks(**kwargs):
    return [OpenMPTestDict(**kwargs)]
```

3.4.3 Running on Multiple Nodes

So far, all our tests run on a single node. Depending on the actual system that ReFrame is running, the test may run locally or be submitted to the system's job scheduler. In this example, we write a regression test for the MPI+OpenMP version of the matrix-vector product. The source code of this program is in `tutorial/src/example_matrix_vector_multiplication_mpi_openmp.c`. The regression test file follows:

```python
import os
import reframe.utility.sanity as sn

from reframe.core.pipeline import RegressionTest


class MPITest(RegressionTest):
    def __init__(self, **kwargs):
        super().__init__('example3_check',
                         os.path.dirname(__file__), **kwargs)
        self.descr = 'Matrix-vector multiplication example with MPI'
        self.valid_systems = ['daint:gpu', 'daint:mc']
        self.valid_prog_environs = ['PrgEnv-cray', 'PrgEnv-gnu',
                                    'PrgEnv-intel', 'PrgEnv-pgi']
        self.sourcepath = 'example_matrix_vector_multiplication_mpi_openmp.c'
```

```python
        self.executable_opts = ['1024', '10']
        self.prgenv_flags = {
            'PrgEnv-cray':  '-homp',
            'PrgEnv-gnu':   '-fopenmp',
            'PrgEnv-intel': '-openmp',
            'PrgEnv-pgi':   '-mp'
        }
        self.sanity_patterns = sn.assert_found(
            r'time for single matrix vector multiplication', self.stdout)
        self.num_tasks = 8
        self.num_tasks_per_node = 2
        self.num_cpus_per_task = 4
        self.variables = {
            'OMP_NUM_THREADS': str(self.num_cpus_per_task)
        }
        self.maintainers = ['you-can-type-your-email-here']
        self.tags = {'tutorial'}

    def compile(self):
        prgenv_flags = self.prgenv_flags[self.current_environ.name]
        self.current_environ.cflags = prgenv_flags
        super().compile()


def _get_checks(**kwargs):
    return [MPITest(**kwargs)]
```

This test is pretty much similar to the *test example* for the OpenMP code we have shown before, except that it adds some information about the configuration of the distributed tasks. It also restricts the valid systems only to those that support distributed execution. Let's take the changes step-by-step:

First we need to specify for which partitions this test is meaningful by setting the *valid_systems* attribute:

```python
self.valid_systems = ['daint:gpu', 'daint:mc']
```

We only specify the partitions that are configured with a job scheduler. If we try to run the generated executable on the login nodes, it will fail. So we remove this partition from the list of the supported systems.

The most important addition to this check are the variables controlling the distributed execution:

```python
self.num_tasks = 8
self.num_tasks_per_node = 2
self.num_cpus_per_task = 4
```

By setting these variables, we specify that this test should run with 8 MPI tasks in total, using two tasks per node. Each task may use four logical CPUs. Based on these variables ReFrame will generate the appropriate scheduler flags to meet that requirement. For example, for Slurm these variables will result in the following flags: `--ntasks=8`, `--ntasks-per-node=2` and `--cpus-per-task=4`. ReFrame provides several more variables for configuring the job submission. As shown in the following Table, they follow closely the corresponding Slurm options. For schedulers that do not provide the same functionality, some of the variables may be ignored.

RegressionTest attribute	Corresponding SLURM option
`time_limit = (0, 10, 30)`	`--time=00:10:30`
`use_multithreading = True`	`--hint=multithread`
`use_multithreading = False`	`--hint=nomultithread`
`exclusive = True`	`--exclusive`
`num_tasks=72`	`--ntasks=72`
`num_tasks_per_node=36`	`--ntasks-per-node=36`
`num_cpus_per_task=4`	`--cpus-per-task=4`
`num_tasks_per_core=2`	`--ntasks-per-core=2`
`num_tasks_per_socket=36`	`--ntasks-per-socket=36`

3.4.4 Testing a GPU Code

In this example, we will create two regression tests for two different GPU versions of our matrix-vector code: OpenACC and CUDA. Let's start with the OpenACC regression test:

```python
import os
import reframe.utility.sanity as sn

from reframe.core.pipeline import RegressionTest


class OpenACCTest(RegressionTest):
    def __init__(self, **kwargs):
        super().__init__('example4_check',
                         os.path.dirname(__file__), **kwargs)
        self.descr = 'Matrix-vector multiplication example with OpenACC'
        self.valid_systems = ['daint:gpu']
        self.valid_prog_environs = ['PrgEnv-cray', 'PrgEnv-pgi']
        self.sourcepath = 'example_matrix_vector_multiplication_openacc.c'
        self.executable_opts = ['1024', '100']
        self.modules = ['craype-accel-nvidia60']
        self.num_gpus_per_node = 1
        self.prgenv_flags = {
            'PrgEnv-cray': '-hacc -hnoomp',
            'PrgEnv-pgi':  '-acc -ta=tesla:cc60'
        }
        self.sanity_patterns = sn.assert_found(
            r'time for single matrix vector multiplication', self.stdout)
        self.maintainers = ['you-can-type-your-email-here']
        self.tags = {'tutorial'}

    def compile(self):
        prgenv_flags = self.prgenv_flags[self.current_environ.name]
        self.current_environ.cflags = prgenv_flags
        super().compile()


def _get_checks(**kwargs):
    return [OpenACCTest(**kwargs)]
```

The things to notice in this test are the restricted list of system partitions and programming environments that this test supports and the use of the *modules* variable:

```
self.modules = ['craype-accel-nvidia60']
```

The *modules* variable takes a list of modules that should be loaded during the setup phase of the test. In this particular test, we need to load the `craype-accel-nvidia60` module, which enables the generation of a GPU binary from an OpenACC code.

It is also important to note that in GPU-enabled tests the number of GPUs for each node have to be specified by setting the corresponding variable *num_gpus_per_node*, as follows:

```
self.num_gpus_per_node = 1
```

The regression test for the CUDA code is slightly simpler:

```python
import os
import reframe.utility.sanity as sn

from reframe.core.pipeline import RegressionTest


class CudaTest(RegressionTest):
    def __init__(self, **kwargs):
        super().__init__('example5_check',
                         os.path.dirname(__file__), **kwargs)
        self.descr = 'Matrix-vector multiplication example with CUDA'
        self.valid_systems = ['daint:gpu']
        self.valid_prog_environs = ['PrgEnv-cray', 'PrgEnv-gnu', 'PrgEnv-pgi']
        self.sourcepath = 'example_matrix_vector_multiplication_cuda.cu'
        self.executable_opts = ['1024', '100']
        self.modules = ['cudatoolkit']
        self.num_gpus_per_node = 1
        self.sanity_patterns = sn.assert_found(
            r'time for single matrix vector multiplication', self.stdout)
        self.maintainers = ['you-can-type-your-email-here']
        self.tags = {'tutorial'}


def _get_checks(**kwargs):
    return [CudaTest(**kwargs)]
```

ReFrame will recognize the `.cu` extension of the source file and it will try to invoke `nvcc` for compiling the code. In this case, there is no need to differentiate across the programming environments, since the compiler will be eventually the same. `nvcc` in our example is provided by the `cudatoolkit` module, which we list it in the *modules* variable.

3.4.5 More Advanced Sanity Checking

So far we have done a very simple sanity checking. We are only looking if a specific line is present in the output of the test program. In this example, we expand the regression test of the serial code, so as to check also if the printed norm of the result matrix is correct.

```python
import os
import reframe.utility.sanity as sn

from reframe.core.pipeline import RegressionTest


class SerialNormTest(RegressionTest):
```

```python
    def __init__(self, **kwargs):
        super().__init__('example6_check',
                         os.path.dirname(__file__), **kwargs)
        self.descr = 'Matrix-vector multiplication with L2 norm check'
        self.valid_systems = ['*']
        self.valid_prog_environs = ['*']
        self.sourcepath = 'example_matrix_vector_multiplication.c'

        matrix_dim = 1024
        iterations = 100
        self.executable_opts = [str(matrix_dim), str(iterations)]

        expected_norm = matrix_dim
        found_norm = sn.extractsingle(
            r'The L2 norm of the resulting vector is:\s+(?P<norm>\S+)',
            self.stdout, 'norm', float)
        self.sanity_patterns = sn.all([
            sn.assert_found(
                r'time for single matrix vector multiplication', self.stdout),
            sn.assert_lt(sn.abs(expected_norm - found_norm), 1.0e-6)
        ])
        self.maintainers = ['you-can-type-your-email-here']
        self.tags = {'tutorial'}


def _get_checks(**kwargs):
    return [SerialNormTest(**kwargs)]
```

The only difference with our first example is actually the more complex expression to assess the sanity of the test. Let's go over it line-by-line. The first thing we do is to extract the norm printed in the standard output.

```python
found_norm = sn.extractsingle(
    r'The L2 norm of the resulting vector is:\s+(?P<norm>\S+)',
    self.stdout, 'norm', float)
```

The *extractsingle* sanity function extracts some information from a single occurrence (by default the first) of a pattern in a filename. In our case, this function will extract the norm capturing group from the match of the regular expression `r'The L2 norm of the resulting vector is:\s+(?P<norm>\S+)'` in standard output, it will convert it to float and it will return it. Unnamed capturing groups in regular expressions are also supported, which you can reference by their group number. For example, we could have written the same statement as follows:

```python
found_norm = sn.extractsingle(
    r'The L2 norm of the resulting vector is:\s+(\S+)',
    self.stdout, 1, float)
```

Notice that we replaced the `'norm'` argument with `1`, which is the capturing group number.

Note: In regular expressions, capturing group `0` corresponds always to the whole match. In sanity functions dealing with regular expressions, this will yield the whole line that matched.

A useful counterpart of *extractsingle* is the *extractall* function, which instead of a single occurrence, returns a list of all the occurrences found. For a more detailed description of this and other sanity functions, please refer to the sanity function reference.

The next couple of lines is the actual sanity check:

```
self.sanity_patterns = sn.all([
    sn.assert_found(
        r'time for single matrix vector multiplication', self.stdout),
    sn.assert_lt(sn.abs(expected_norm - found_norm), 1.0e-6)
])
```

This expression combines two conditions that need to true, in order for the sanity check to succeed:

1. Find in standard output the same line we were looking for already in the first example.

2. Verify that the printed norm does not deviate significantly from the expected value.

The *all* function is responsible for combining the results of the individual subexpressions. It is essentially the Python built-in all() function, exposed as a sanity function, and requires that all the elements of the iterable it takes as an argument evaluate to True. As mentioned before, all the assert_* functions either return True on success or raise SanityError. So, if everything goes smoothly, sn.all() will evaluate to True and sanity checking will succeed.

The expression for the second condition is more interesting. Here, we want to assert that the absolute value of the difference between the expected and the found norm are below a certain value. The important thing to mention here is that you can combine the results of sanity functions in arbitrary expressions, use them as arguments to other functions, return them from functions, assign them to variables etc. Remember that sanity functions are not evaluated at the time you call them. They will be evaluated later by the framework during the sanity checking phase. If you include the result of a sanity function in an expression, the evaluation of the resulting expression will also be deferred. For a detailed description of the mechanism behind the sanity functions, please have a look at "Understanding The Mechanism Of Sanity Functions" section.

3.4.6 Writing a Performance Test

An important aspect of regression testing is checking for performance regressions. ReFrame offers a flexible way of extracting and manipulating performance data from the program output, as well as a comprehensive way of setting performance thresholds per system and system partitions.

In this example, we extend the CUDA test presented previously, so as to check also the performance of the matrix-vector multiplication.

```
import os
import reframe.utility.sanity as sn

from reframe.core.pipeline import RegressionTest


class CudaPerfTest(RegressionTest):
    def __init__(self, **kwargs):
        super().__init__('example7_check',
                         os.path.dirname(__file__), **kwargs)
        self.descr = 'Matrix-vector multiplication (CUDA performance test)'
        self.valid_systems = ['daint:gpu']
        self.valid_prog_environs = ['PrgEnv-gnu', 'PrgEnv-cray', 'PrgEnv-pgi']
        self.sourcepath = 'example_matrix_vector_multiplication_cuda.cu'
        self.executable_opts = ['4096', '1000']
        self.modules = ['cudatoolkit']
        self.num_gpus_per_node = 1
        self.sanity_patterns = sn.assert_found(
            r'time for single matrix vector multiplication', self.stdout)
        self.perf_patterns = {
            'perf': sn.extractsingle(r'Performance:\s+(?P<Gflops>\S+) Gflop/s',
```

```
                                      self.stdout, 'Gflops', float)
        }
        self.reference = {
            'daint:gpu': {
                'perf': (50.0, -0.1, 0.1),
            }
        }
        self.maintainers = ['you-can-type-your-email-here']
        self.tags = {'tutorial'}

    def compile(self):
        self.current_environ.cxxflags = '-O3'
        super().compile()


def _get_checks(**kwargs):
    return [CudaPerfTest(**kwargs)]
```

The are two new variables set in this test that basically enable the performance testing:

_perf_patterns_ This variable defines which are the performance patterns we are looking for and how to extract the performance values.

reference This variable is a collection of reference values for different systems.

Let's have a closer look at each of them:

```
self.perf_patterns = {
    'perf': sn.extractsingle(r'Performance:\s+(?P<Gflops>\S+) Gflop/s',
                             self.stdout, 'Gflops', float)
}
```

The *perf_patterns* attribute is a dictionary, whose keys are *performance variables* (i.e., arbitrary names assigned to the performance values we are looking for), and its values are *sanity expressions* that specify how to obtain these performance values from the output. A sanity expression is a Python expression that uses the result of one or more *sanity functions*. In our example, we name the performance value we are looking for simply as perf and we extract its value by converting to float the regex capturing group named Gflops from the line that was matched in the standard output.

Each of the performance variables defined in *perf_patterns* must be resolved in the *reference* dictionary of reference values. When the framework obtains a performance value from the output of the test it searches for a reference value in the *reference* dictionary, and then it checks whether the user supplied tolerance is respected. Let's go over the *reference* dictionary of our example and explain its syntax in more detail:

```
self.reference = {
    'daint:gpu': {
        'perf': (50.0, -0.1, 0.1),
    }
}
```

This is a special type of dictionary that we call scoped dictionary, because it defines scopes for its keys. We have already seen it being used in the environments section of the configuration file of ReFrame. In order to resolve a reference value for a performance variable, ReFrame creates the following key <current_sys>:<current_part>:<perf_variable> and looks it up inside the *reference* dictionary. If our example, since this test is only allowed to run on the daint:gpu partition of our system, ReFrame will look for the daint:gpu:perf reference key. The perf subkey will then be searched in the following scopes in this order: daint:gpu, daint, *. The first occurrence will be used as the reference value of the perf performance

---

variable. In our example, the `perf` key will be resolved in the `daint:gpu` scope giving us the reference value.

Reference values in ReFrame are specified as a three-tuple comprising the reference value and lower and upper thresholds. Thresholds are specified as decimal fractions of the reference value. The lower threshold must lie in the [-1,0] interval, whereas the upper threshold must be lie in the [0,1] interval. In our example, the reference value for this test on `daint:gpu` is 50 Gflop/s ±10%. Setting a threshold value to `None` disables the threshold.

3.4.7 Combining It All Together

As we have mentioned before and as you have already experienced with the examples in this tutorial, regression tests in ReFrame are written in pure Python. As a result, you can leverage the language features and capabilities to organize better your tests and decrease the maintenance cost. In this example, we are going to reimplement all the tests of the tutorial with much less code and in a single file. Here is the final example code that combines all the tests discussed before:

```python
import os
import reframe.utility.sanity as sn

from reframe.core.pipeline import RegressionTest


class BaseMatrixVectorTest(RegressionTest):
    def __init__(self, test_version, **kwargs):
        super().__init__('example8_' + test_version.lower() + '_check',
                         os.path.dirname(__file__), **kwargs)
        self.descr = '%s matrix-vector multiplication' % test_version
        self.valid_systems = ['*']
        self.valid_prog_environs = ['*']
        self.prgenv_flags = None

        matrix_dim = 1024
        iterations = 100
        self.executable_opts = [str(matrix_dim), str(iterations)]

        expected_norm = matrix_dim
        found_norm = sn.extractsingle(
            r'The L2 norm of the resulting vector is:\s+(?P<norm>\S+)',
            self.stdout, 'norm', float)
        self.sanity_patterns = sn.all([
            sn.assert_found(
                r'time for single matrix vector multiplication', self.stdout),
            sn.assert_lt(sn.abs(expected_norm - found_norm), 1.0e-6)
        ])
        self.maintainers = ['you-can-type-your-email-here']
        self.tags = {'tutorial'}


    def compile(self):
        if self.prgenv_flags is not None:
            self.current_environ.cflags = self.prgenv_flags[self.current_environ.name]

        super().compile()


class SerialTest(BaseMatrixVectorTest):
    def __init__(self, **kwargs):
```

(continues on next page)

```python
        super().__init__('Serial', **kwargs)
        self.sourcepath = 'example_matrix_vector_multiplication.c'


class OpenMPTest(BaseMatrixVectorTest):
    def __init__(self, **kwargs):
        super().__init__('OpenMP', **kwargs)
        self.sourcepath = 'example_matrix_vector_multiplication_openmp.c'
        self.valid_prog_environs = ['PrgEnv-cray', 'PrgEnv-gnu',
                                    'PrgEnv-intel', 'PrgEnv-pgi']
        self.prgenv_flags = {
            'PrgEnv-cray':  '-homp',
            'PrgEnv-gnu':   '-fopenmp',
            'PrgEnv-intel': '-openmp',
            'PrgEnv-pgi':   '-mp'
        }
        self.variables = {
            'OMP_NUM_THREADS': '4'
        }


class MPITest(BaseMatrixVectorTest):
    def __init__(self, **kwargs):
        super().__init__('MPI', **kwargs)
        self.valid_systems = ['daint:gpu', 'daint:mc']
        self.valid_prog_environs = ['PrgEnv-cray', 'PrgEnv-gnu',
                                    'PrgEnv-intel', 'PrgEnv-pgi']
        self.sourcepath = 'example_matrix_vector_multiplication_mpi_openmp.c'
        self.prgenv_flags = {
            'PrgEnv-cray':  '-homp',
            'PrgEnv-gnu':   '-fopenmp',
            'PrgEnv-intel': '-openmp',
            'PrgEnv-pgi':   '-mp'
        }
        self.num_tasks = 8
        self.num_tasks_per_node = 2
        self.num_cpus_per_task = 4
        self.variables = {
            'OMP_NUM_THREADS': str(self.num_cpus_per_task)
        }


class OpenACCTest(BaseMatrixVectorTest):
    def __init__(self, **kwargs):
        super().__init__('OpenACC', **kwargs)
        self.valid_systems = ['daint:gpu']
        self.valid_prog_environs = ['PrgEnv-cray', 'PrgEnv-pgi']
        self.sourcepath = 'example_matrix_vector_multiplication_openacc.c'
        self.modules = ['craype-accel-nvidia60']
        self.num_gpus_per_node = 1
        self.prgenv_flags = {
            'PrgEnv-cray': '-hacc -hnoomp',
            'PrgEnv-pgi':  '-acc -ta=tesla:cc60'
        }


class CudaTest(BaseMatrixVectorTest):
```

```python
    def __init__(self, **kwargs):
        super().__init__('CUDA', **kwargs)
        self.valid_systems = ['daint:gpu']
        self.valid_prog_environs = ['PrgEnv-gnu', 'PrgEnv-cray', 'PrgEnv-pgi']
        self.sourcepath = 'example_matrix_vector_multiplication_cuda.cu'
        self.modules = ['cudatoolkit']
        self.num_gpus_per_node = 1


def _get_checks(**kwargs):
    return [SerialTest(**kwargs), OpenMPTest(**kwargs), MPITest(**kwargs),
            OpenACCTest(**kwargs), CudaTest(**kwargs)]
```

This test abstracts away the common functionality found in almost all of our tutorial tests (executable options, sanity checking, etc.) to a base class, from which all the concrete regression tests derive. Each test then redefines only the parts that are specific to it. The _get_checks() now instantiates all the interesting tests and returns them as a list to the framework. The total line count of this refactored example is less than half of that of the individual tutorial tests. Notice how the base class for all tutorial regression tests specify additional parameters to its constructor, so that the concrete subclasses can initialize it based on their needs.

Another interesting technique, not demonstrated here, is to create regression test factories that will create different regression tests based on specific arguments they take in their constructor.

We use such techniques extensively in the regression tests for our production systems, in order to facilitate their maintenance.

3.4.8 Summary

This concludes our ReFrame tutorial. We have covered all basic aspects of writing regression tests in ReFrame and you should now be able to start experimenting by writing your first useful tests. The next section covers further topics in customizing a regression test to your needs.

3.5 Customizing Further a Regression Test

In this section, we are going to show some more elaborate use cases of ReFrame. Through the use of more advanced examples, we will demonstrate further customization options which modify the default options of the ReFrame pipeline. The corresponding scripts as well as the source code of the examples discussed here can be found in the directory `tutorial/advanced`.

3.5.1 Leveraging Makefiles

We have already shown how you can compile a single source file associated with your regression test. In this example, we show how ReFrame can leverage Makefiles to build executables.

Compiling a regression test through a Makefile is very straightforward with ReFrame. If the *sourcepath* attribute refers to a directory, then ReFrame will automatically invoke `make` there.

Note: More specifically, ReFrame will compile the source files found in the directory that is constructed as `os.path.join(self.sourcesdir, self.sourcepath)` (given that `sourcesdir` is defined, as it is usually the case).

If `sourcesdir` is not defined, ReFrame assumes that the user will make sure that the source files will be in the stagedir (`_stagedir`) at the moment of compilation (more information on the stagedir directory is found in "Running ReFrame" section). The user may for instance generate the source files or copy them from a git repository by the means of some commands defined in `prebuild_cmd`. Thus, ReFrame will in this case compile the sources files found in the directory that is constructed as `os.path.join(self._stagedir, self.sourcepath)`.

By default, *sourcepath* is the empty string and *sourcesdir* is set `src/`. As a result, by not specifying a *sourcepath* at all, ReFrame will try to invoke `make` inside the `src/` directory of the test. This is exactly what our first example here does.

For completeness, here are the contents of `Makefile` provided:

```
EXECUTABLE := advanced_example1

OBJS := advanced_example1.o

$(EXECUTABLE): $(OBJS)
	$(CC) $(CFLAGS) $(LDFLAGS) -o $@ $^

$(OBJS): advanced_example1.c
	$(CC) $(CPPFLAGS) $(CFLAGS) -c $(LDFLAGS) -o $@ $^
```

The corresponding `advanced_example1.c` source file consists of a simple printing of a message, whose content depends on the preprocessor variable `MESSAGE`:

```
#include <stdio.h>

int main(){
#ifdef MESSAGE
	char *message = "SUCCESS";
#else
	char *message = "FAILURE";
#endif
	printf("Setting of preprocessor variable: %s\n", message);
	return 0;
}
```

The purpose of the regression test in this case is to set the preprocessor variable `MESSAGE` via `CPPFLAGS` and then check the standard output for the message `SUCCESS`, which indicates that the preprocessor flag has been passed and processed correctly by the Makefile.

The contents of this regression test are the following (`tutorial/advanced/advanced_example1.py`):

```
import os

import reframe.utility.sanity as sn
from reframe.core.pipeline import RegressionTest


class MakefileTest(RegressionTest):
    def __init__(self, **kwargs):
        super().__init__('preprocessor_check', os.path.dirname(__file__),
                         **kwargs)

        self.descr = ('ReFrame tutorial demonstrating the use of Makefiles '
                      'and compile options')
        self.valid_systems = ['*']
        self.valid_prog_environs = ['*']
```

(continues on next page)

```
        self.executable = './advanced_example1'
        self.sanity_patterns = sn.assert_found('SUCCESS', self.stdout)
        self.maintainers = ['put-your-name-here']
        self.tags = {'tutorial'}

    def compile(self):
        self.current_environ.cppflags = '-DMESSAGE'
        super().compile()


def _get_checks(**kwargs):
    return [MakefileTest(**kwargs)]
```

The important bit here is the `compile()` method.

```
def compile(self):
    self.current_environ.cppflags = '-DMESSAGE'
    super().compile()
```

As in the simple single source file examples we showed in the tutorial, we use the current programming environment's flags for modifying the compilation. ReFrame will then compile the regression test source code as by invoking `make` as follows:

```
make CC=cc CXX=CC FC=ftn CPPFLAGS=-DMESSAGE
```

Notice, how ReFrame passes all the programming environment's variables to the `make` invocation. It is important to note here that, if a set of flags is set to `None` (the default, if not otherwise set in the ReFrame's configuration), these are not passed to `make`. You can also completely disable the propagation of any flags to `make` by setting *self.propagate = False* in your regression test.

At this point it is useful also to note that you can also use a custom Makefile, not named `Makefile` or after any other standard Makefile name. In this case, you can pass the custom Makefile name as an argument to the compile method of the base *RegressionTest* class as follows:

```
super().compile(makefile='Makefile_custom')
```

3.5.2 Implementing a Run-Only Regression Test

There are cases when it is desirable to perform regression testing for an already built executable. The following test uses the `echo` Bash shell command to print a random integer between specific lower and upper bounds. Here is the full regression test (`tutorial/advanced/advanced_example2.py`):

```
import os

import reframe.utility.sanity as sn
from reframe.core.pipeline import RunOnlyRegressionTest


class RunOnlyTest(RunOnlyRegressionTest):
    def __init__(self, **kwargs):
        super().__init__('run_only_check', os.path.dirname(__file__),
                         **kwargs)

        self.descr = ('ReFrame tutorial demonstrating the class'
```

```
                     'RunOnlyRegressionTest')
        self.valid_systems = ['*']
        self.valid_prog_environs = ['*']
        self.sourcesdir = None

        lower = 90
        upper = 100
        self.executable = 'echo "Random: $((RANDOM%({1}+1-{0})+{0}))"'.format(
            lower, upper)
        self.sanity_patterns = sn.assert_bounded(sn.extractsingle(
            r'Random: (?P<number>\S+)', self.stdout, 'number', float),
            lower, upper)
        self.maintainers = ['put-your-name-here']
        self.tags = {'tutorial'}


def _get_checks(**kwargs):
    return [RunOnlyTest(**kwargs)]
```

There is nothing special for this test compared to those presented earlier except that it derives from the
RunOnlyRegressionTest. A thing to note about run-only regression tests is that the copying of their resources
to the stage directory is performed at the beginning of the run phase. For standard regression tests, this happens at the
beginning of the compilation phase, instead. Furthermore, in this particular test the *executable* consists only of
standard Bash shell commands. For this reason, we can set *sourcesdir* to None informing ReFrame that the test
does not have any resources.

3.5.3 Implementing a Compile-Only Regression Test

ReFrame provides the option to write compile-only tests which consist only of a compilation phase without a spec-
ified executable. This kind of tests must derive from the *CompileOnlyRegressionTest* class provided by the
framework. The following example (tutorial/advanced/advanced_example3.py) reuses the code of our
first example in this section and checks that no warnings are issued by the compiler:

```
import os

import reframe.utility.sanity as sn
from reframe.core.pipeline import CompileOnlyRegressionTest


class CompileOnlyTest(CompileOnlyRegressionTest):
    def __init__(self, **kwargs):
        super().__init__('compile_only_check', os.path.dirname(__file__),
                         **kwargs)
        self.descr = ('ReFrame tutorial demonstrating the class'
                      'CompileOnlyRegressionTest')
        self.valid_systems = ['*']
        self.valid_prog_environs = ['*']
        self.sanity_patterns = sn.assert_not_found('warning', self.stderr)

        self.maintainers = ['put-your-name-here']
        self.tags = {'tutorial'}


def _get_checks(**kwargs):
    return [CompileOnlyTest(**kwargs)]
```

The important thing to note here is that the standard output and standard error of the tests, accessible through the *stdout* and *stderr* attributes, are now the corresponding those of the compilation command. So sanity checking can be done in exactly the same way as with a normal test.

3.5.4 Leveraging Environment Variables

We have already demonstrated in the tutorial that ReFrame allows you to load the required modules for regression tests and also set any needed environment variables. When setting environment variables for your test through the *variables* attribute, you can assign them values of other, already defined, environment variables using the standard notation $OTHER_VARIABLE or ${OTHER_VARIABLE}. The following regression test (tutorial/advanced/advanced_example4.py) sets the CUDA_HOME environment variable to the value of the CUDATOOLKIT_HOME and then compiles and runs a simple program:

```python
import os

import reframe.utility.sanity as sn
from reframe.core.pipeline import RegressionTest


class EnvironmentVariableTest(RegressionTest):
    def __init__(self, **kwargs):
        super().__init__('env_variable_check', os.path.dirname(__file__),
                         **kwargs)

        self.descr = ('ReFrame tutorial demonstrating the use'
                      'of environment variables provided by loaded modules')
        self.valid_systems = ['daint:gpu']
        self.valid_prog_environs = ['*']
        self.modules = ['cudatoolkit']
        self.variables = {'CUDA_HOME': '$CUDATOOLKIT_HOME'}
        self.executable = './advanced_example4'
        self.sanity_patterns = sn.assert_found(r'SUCCESS', self.stdout)
        self.maintainers = ['put-your-name-here']
        self.tags = {'tutorial'}

    def compile(self):
        super().compile(makefile='Makefile_example4')


def _get_checks(**kwargs):
    return [EnvironmentVariableTest(**kwargs)]
```

Before discussing this test in more detail, let's first have a look in the source code and the Makefile of this example:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#ifndef CUDA_HOME
#   define CUDA_HOME ""
#endif

int main() {
    char *cuda_home_compile = CUDA_HOME;
    char *cuda_home_runtime = getenv("CUDA_HOME");
    if (cuda_home_runtime &&
```

(continues on next page)

```
        strnlen(cuda_home_runtime, 256) &&
        strnlen(cuda_home_compile, 256) &&
        !strncmp(cuda_home_compile, cuda_home_runtime, 256)) {
        printf("SUCCESS\n");
    } else {
        printf("FAILURE\n");
        printf("Compiled with CUDA_HOME=%s, ran with CUDA_HOME=%s\n",
               cuda_home_compile,
               cuda_home_runtime ? cuda_home_runtime : "<null>");
    }

    return 0;
}
```

This program is pretty basic, but enough to demonstrate the use of environment variables from ReFrame. It simply compares the value of the CUDA_HOME macro with the value of the environment variable CUDA_HOME at runtime, printing SUCCESS if they are not empty and match. The Makefile for this example compiles this source by simply setting CUDA_HOME to the value of the CUDA_HOME environment variable:

```
EXECUTABLE := advanced_example4

CPPFLAGS = -DCUDA_HOME=\"$(CUDA_HOME)\"

.SUFFIXES: .o .c

OBJS := advanced_example4.o

$(EXECUTABLE): $(OBJS)
	$(CC) $(CFLAGS) $(LDFLAGS) -o $@ $^

$(OBJS): advanced_example4.c
	$(CC) $(CPPFLAGS) $(CFLAGS) -c $(LDFLAGS) -o $@ $^

clean:
	/bin/rm -f $(OBJS) $(EXECUTABLE)
```

Coming back now to the ReFrame regression test, the CUDATOOLKIT_HOME environment variable is defined by the cudatoolkit module. If you try to run the test, you will see that it will succeed, meaning that the CUDA_HOME variable was set correctly both during the compilation and the runtime.

When ReFrame sets up a test, it first loads its required modules and then sets the required environment variables expanding their values. This has the result that CUDA_HOME takes the correct value in our example at the compilation time.

At runtime, ReFrame will generate the following instructions in the shell script associated with this test:

```
module load cudatoolkit
export CUDA_HOME=$CUDATOOLKIT_HOME
```

This ensures that the environment of the test is also set correctly at runtime.

Finally, as already mentioned *previously*, since the Makefile name is not one of the standard ones, it has to be passed as an argument to the *compile* method of the base *RegressionTest* class as follows:

```
super().compile(makefile='Makefile_example4')
```

3.5.5 Setting a Time Limit for Regression Tests

ReFrame gives you the option to limit the execution time of regression tests. The following example (`tutorial/advanced/advanced_example5.py`) demonstrates how you can achieve this by limiting the execution time of a test that tries to sleep 100 seconds:

```python
import os

import reframe.utility.sanity as sn
from reframe.core.pipeline import RunOnlyRegressionTest


class TimeLimitTest(RunOnlyRegressionTest):
    def __init__(self, **kwargs):
        super().__init__('time_limit_check', os.path.dirname(__file__),
                         **kwargs)

        self.descr = ('ReFrame tutorial demonstrating the use'
                      'of a user-defined time limit')
        self.valid_systems = ['daint:gpu', 'daint:mc']
        self.valid_prog_environs = ['*']
        self.time_limit = (0, 1, 0)
        self.executable = 'sleep'
        self.executable_opts = ['100']
        self.sanity_patterns = sn.assert_found(
            r'CANCELLED.*DUE TO TIME LIMIT', self.stderr)
        self.maintainers = ['put-your-name-here']
        self.tags = {'tutorial'}


def _get_checks(**kwargs):
    return [TimeLimitTest(**kwargs)]
```

The important bit here is the following line that sets the time limit for the test to one minute:

```python
self.time_limit = (0, 1, 0)
```

The *time_limit* attribute is a three-tuple in the form `(HOURS, MINUTES, SECONDS)`. Time limits are implemented for all the scheduler backends.

The sanity condition for this test verifies that associated job has been canceled due to the time limit.

```python
self.sanity_patterns = sn.assert_found('CANCELLED.*TIME LIMIT', self.stderr)
```

3.5.6 Applying a sanity function iteratively

It is often the case that a common sanity pattern has to be applied many times. In this example we will demonstrate how the above situation can be easily tackled using the *sanity* functions offered by ReFrame. Specifically, we would like to execute the following shell script and check that its output is correct:

```bash
#!/usr/bin/env bash

for i in {1..100}
do
    echo Random: $((RANDOM%($UPPER+1-$LOWER)+$LOWER))
done
```

The above script simply prints 100 random integers between the limits given by the variables LOWER and UPPER. For this example the above limits are exported as environment variables by the limits.sh script as follows:

```
#!/usr/bin/env bash

export LOWER=90
export UPPER=100
```

In the corresponding regression test we want to check that all the random numbers printed lie between 90 and 100 ensuring that the script executed correctly. Hence, a common sanity check has to be applied to all the printed random numbers. In ReFrame this can achieved by the use of *map* sanity function accepting a function and an iterable as arguments. Through *map* the given function will be applied to all the members of the iterable object. Note that since *map* is a sanity function, its execution will be deferred. The contents of the ReFrame regression test contained in advanced_example6.py are the following:

```python
import os

import reframe.utility.sanity as sn
from reframe.core.pipeline import RunOnlyRegressionTest


class DeferredIterationTest(RunOnlyRegressionTest):
    def __init__(self, **kwargs):
        super().__init__('deferred_iteration_check',
                         os.path.dirname(__file__), **kwargs)

        self.descr = ('ReFrame tutorial demonstrating the use of deferred '
                      'iteration via the `map` sanity function.')

        self.valid_systems = ['*']
        self.valid_prog_environs = ['*']

        self.executable = './advanced_example6.sh'
        numbers = sn.extractall(r'Random: (?P<number>\S+)', self.stdout,
                                'number', float)

        self.sanity_patterns = sn.and_(
            sn.assert_eq(sn.count(numbers), 100),
            sn.all(sn.map(lambda x: sn.assert_bounded(x, 90, 100), numbers)))

        self.maintainers = ['put-your-name-here']
        self.tags = {'tutorial'}

    def setup(self, partition, environ, **job_opts):
        super().setup(partition, environ, **job_opts)
        self.job.pre_run = ['source %s/limits.sh' % self.stagedir]


def _get_checks(**kwargs):
    return [DeferredIterationTest(**kwargs)]
```

First the random numbers are extracted through the *extractall* function as follows:

```python
numbers = sn.extractall(r'Random: (?P<number>\S+)', self.stdout,
                        'number', float)
```

The numbers variable is a deferred iterable, which upon evaluation will return all the extracted numbers. In order to check that the extracted numbers lie within the specified limits, we make use of the *map* sanity function, which will

apply the *assert_bounded* to all the elements of `numbers`. Additionally, our requirement is that all the numbers satisfy the above constraint and we therefore use *all*.

There is still a small complication that needs to be addressed. The *all* function returns `True` for empty iterables, which is not what we want. So we must ensure that all the numbers are extracted as well. To achieve this, we make use of *count* to get the number of elements contained in `numbers` combined with *assert_eq* to check that the number is indeed 100. Finally, both of the above conditions have to be satisfied for the program execution to be considered successful, hence the use of the *and_* function. Note that the `and` operator is not deferrable and will trigger the evaluation of any deferrable argument passed to it.

The full syntax for the `sanity_patterns` is the following:

```
self.sanity_patterns = sn.and_(
    sn.assert_eq(sn.count(numbers), 100),
    sn.all(sn.map(lambda x: sn.assert_bounded(x, 90, 100), numbers)))
```

Note that the environment variables `LOWER` and `UPPER` have to be exported before execution of the `advanced_example6.sh` script. Within ReFrame it is possible to define commands that will be run before execution of the actual *executable*. To achieve this, the *setup* method has to be overriden to access the *pre_run* field of the corresponding job. In this particular case, the setup implementation is written as:

```
def setup(self, partition, environ, **job_opts):
    super().setup(partition, environ, **job_opts)
    self.job.pre_run = ['source %s/limits.sh' % self.stagedir]
```

The *pre_run* attribute is a list of shell commands to be emitted verbatim in the generated job script before the executable. In this case, we make sure that the `limits.sh` file is sourced before executing the `advanced_example6.sh` script.

3.6 Understanding the Mechanism of Sanity Functions

This section describes the mechanism behind the sanity functions that are used for the sanity and performance checking. Generally, writing a new sanity function is as straightforward as decorating a simple Python function with either the *sanity_function* or the `@reframe.core.deferrable.deferrable` decorator. However, it is important to understand how and when a deferrable function is evaluated, especially if your function takes as arguments the results of other deferrable functions.

3.6.1 What Is a Deferrable Function?

A deferrable function is a function whose a evaluation is deferred to a later point in time. You can define any function as deferrable by adding the *@sanity_funcion* or the `@deferrable` decorator before its definition. The example below demonstrates a simple scenario:

```
import reframe.utility.sanity as sn

@sn.sanity_function
def foo():
    print('hello')
```

If you try to call `foo()`, its code will not execute:

```
>>> foo()
<reframe.core.deferrable._DeferredExpression object at 0x2b70fff23550>
```

Instead, a special object is returned that represents the function whose execution is deferred. Notice the more general *deferred expression* name of this object. We shall see later on why this name is used.

In order to explicitly trigger the execution of `foo()`, you have to call `evaluate` on it:

```
>>> from reframe.core.deferrable import evaluate
>>> evaluate(foo())
hello
```

If the argument passed to `evaluate` is not a deferred expression, it will be simply returned as is.

Deferrable functions may also be combined as we do with normal functions. Let's extend our example with `foo()` accepting an argument and printing it:

```
import reframe.utility.sanity as sn

@sn.sanity_function
def foo(arg):
    print(arg)

@sn.sanity_function
def greetings():
    return 'hello'
```

If we now do `foo(greetings())`, again nothing will be evaluated:

```
>>> foo(greetings())
<reframe.core.deferrable._DeferredExpression object at 0x2b7100e9e978>
```

If we trigger the evaluation of `foo()` as before, we will get expected result:

```
>>> evaluate(foo(greetings()))
hello
```

Notice how the evaluation mechanism goes down the function call graph and returns the expected result. An alternative way to evaluate this expression would be the following:

```
>>> x = foo(greetings())
>>> x.evaluate()
hello
```

As you may have noticed, you can assign a deferred function to a variable and evaluate it later. You may also do `evaluate(x)`, which is equivalent to `x.evaluate()`.

To demonstrate more clearly how the deferred evaluation of a function works, let's consider the following `size3()` deferrable function that simply checks whether an `iterable` passed as argument has three elements inside it:

```
@sn.sanity_function
def size3(iterable):
    return len(iterable) == 3
```

Now let's assume the following example:

```
>>> l = [1, 2]
>>> x = size3(l)
>>> evaluate(x)
False
>>> l += [3]
```

(continues on next page)

```
>>> evaluate(x)
True
```

We first call `size3()` and store its result in `x`. As expected when we evaluate `x`, `False` is returned, since at the time of the evaluation our list has two elements. We later append an element to our list and reevaluate `x` and we get `True`, since at this point the list has three elements.

Note: Deferred functions and expressions may be stored and (re)evaluated at any later point in the program.

An important thing to point out here is that deferrable functions *capture* their arguments at the point they are called. If you change the binding of a variable name (either explicitly or implicitly by applying an operator to an immutable object), this change will not be reflected when you evaluate the deferred function. The function instead will operate on its captured arguments. We will demonstrate this by replacing the list in the above example with a tuple:

```
>>> l = (1, 2)
>>> x = size3(l)
>>> l += (3,)
>>> l
(1, 2, 3)
>>> evaluate(x)
False
```

Why this is happening? This is because tuples are immutable so when we are doing `l += (3,)` to append to our tuple, Python constructs a new tuple and rebinds `l` to the newly created tuple that has three elements. However, when we called our deferrable function, `l` was pointing to a different tuple object, and that was the actual tuple argument that our deferrable function has captured.

The following augmented example demonstrates this:

```
>>> l = (1, 2)
>>> x = size3(l)
>>> l += (3,)
>>> l
(1, 2, 3)
>>> evaluate(x)
False
>>> l = (1, 2)
>>> id(l)
47764346657160
>>> x = size3(l)
>>> l += (3,)
>>> id(l)
47764330582232
>>> l
(1, 2, 3)
>>> evaluate(x)
False
```

Notice the different IDs of `l` before and after the `+=` operation. This a key trait of deferrable functions and expressions that you should be aware of.

3.6.2 Deferred expressions

You might be still wondering why the internal name of a deferred function refers to the more general term deferred expression. Here is why:

```
>>> @sn.sanity_function
... def size(iterable):
...     return len(iterable)
...
>>> l = [1, 2]
>>> x = 2*(size(l) + 3)
>>> x
<reframe.core.deferrable._DeferredExpression object at 0x2b1288f4e940>
>>> evaluate(x)
10
```

As you can see, you can use the result of a deferred function inside arithmetic operations. The result will be another deferred expression that you can evaluate later. You can practically use any Python builtin operator or builtin function with a deferred expression and the result will be another deferred expression. This is quite a powerful mechanism, since with the standard syntax you can create arbitrary expressions that may be evaluated later in your program.

There are some exceptions to this rule, though. The logical `and`, `or` and `not` operators as well as the `in` operator cannot be deferred automatically. These operators try to take the truthy value of their arguments by calling `bool` on them. As we shall see later, applying the `bool` function on a deferred expression causes its immediate evaluation and returns the result. If you want to defer the execution of such operators, you should use the corresponding *and_*, *or_*, *not_* and *contains* functions in *reframe.utility.sanity*, which basically wrap the expression in a deferrable function.

In summary deferrable functions have the following characteristics:

- You can make any function deferrable by preceding it with the *@sanity_function* or the `@deferrable` decorator.

- When you call a deferrable function, its body is not executed but its arguments are *captured* and an object representing the deferred function is returned.

- You can execute the body of a deferrable function at any later point by calling `evaluate` on the deferred expression object that it has been returned by the call to the deferred function.

- Deferred functions can accept other deferred expressions as arguments and may also return a deferred expression.

- When you evaluate a deferrable function, any other deferrable function down the call tree will also be evaluated.

- You can include a call to a deferrable function in any Python expression and the result will be another deferred expression.

3.6.3 How a Deferred Expression Is Evaluated?

As discussed before, you can create a new deferred expression by calling a function whose definition is decorated by the `@sanity_function` or `@deferrable` decorator or by including an already deferred expression in any sort of arithmetic operation. When you call `evaluate` on a deferred expression, you trigger the evaluation of the whole subexpression tree. Here is how the evaluation process evolves:

A deferred expression object is merely a placeholder of the target function and its arguments at the moment you call it. Deferred expressions leverage also the Python's data model so as to capture all the binary and unary operators supported by the language. When you call `evaluate()` on a deferred expression object, the stored function will be called passing it the captured arguments. If any of the arguments is a deferred expression, it will be evaluated too. If the return value of the deferred expression is also a deferred expression, it will be evaluated as well.

This last property lets you call other deferrable functions from inside a deferrable function. Here is an example where we define two deferrable variations of the builtins sum and `len` and another deferrable function `avg()` that computes the average value of the elements of an iterable by calling our deferred builtin alternatives.

```
@sn.sanity_function
def dsum(iterable):
    return sum(iterable)


@sn.sanity_function
def dlen(iterable):
    return len(iterable)


@sn.sanity_function
def avg(iterable):
    return dsum(iterable) / dlen(iterable)
```

If you try to evaluate `avg()` with a list, you will get the expected result:

```
>>> avg([1, 2, 3, 4])
<reframe.core.deferrable._DeferredExpression object at 0x2b1288f54b70>
>>> evaluate(avg([1, 2, 3, 4]))
2.5
```

The return value of `evaluate(avg())` would normally be a deferred expression representing the division of the results of the other two deferrable functions. However, the evaluation mechanism detects that the return value is a deferred expression and it automatically triggers its evaluation, yielding the expected result. The following figure shows how the evaluation evolves for this particular example:

Fig. 2: Sequence diagram of the evaluation of the deferrable `avg()` function.

3.6.4 Implicit evaluation of a deferred expression

Although you can trigger the evaluation of a deferred expression at any time by calling `evaluate`, there are some cases where the evaluation is triggered implicitly:

- When you try to get the truthy value of a deferred expression by calling `bool` on it. This happens for example when you include a deferred expression in an `if` statement or as an argument to the `and`, `or`, `not` and `in` (`__contains__`) operators. The following example demonstrates this behavior:

  ```
  >>> if avg([1, 2, 3, 4]) > 2:
  ...     print('hello')
  ...
  hello
  ```

  The expression `avg([1, 2, 3, 4]) > 2` is a deferred expression, but its evaluation is triggered from the Python interpreter by calling the `bool()` method on it, in order to evaluate the `if` statement. A similar example is the following that demonstrates the behaviour of the `in` operator:

  ```
  >>> from reframe.core.deferrable import make_deferrable
  >>> l = make_deferrable([1, 2, 3])
  >>> l
  <reframe.core.deferrable._DeferredExpression object at 0x2b1288f54cf8>
  >>> evaluate(l)
  [1, 2, 3]
  ```

(continues on next page)

```
>>> 4 in l
False
>>> 3 in l
True
```

The `make_deferrable` is simply a deferrable version of the identity function (a function that simply returns its argument). As expected, `l` is a deferred expression that evaluates to the `[1, 2, 3]` list. When we apply the `in` operator, the deferred expression is immediately evaluated.

Note: Python expands this expression into `bool(l.__contains__(3))`. Although `__contains__` is also defined as a deferrable function in `_DeferredExpression`, its evaluation is triggered by the `bool` builtin.

- When you try to iterate over a deferred expression by calling the `iter` function on it. This call happens implicitly by the Python interpreter when you try to iterate over a container. Here is an example:

```
>>> @sn.sanity_function
... def getlist(iterable):
...     ret = list(iterable)
...     ret += [1, 2, 3]
...     return ret
>>> getlist([1, 2, 3])
<reframe.core.deferrable._DeferredExpression object at 0x2b1288f54dd8>
>>> for x in getlist([1, 2, 3]):
...     print(x)
...
1
2
3
1
2
3
```

Simply calling `getlist()` will not execute anything and a deferred expression object will be returned. However, when you try to iterate over the result of this call, then the deferred expression will be evaluated immediately.

- When you try to call `str` on a deferred expression. This will be called by the Python interpreter every time you try to print this expression. Here is an example with the `getlist` deferrable function:

```
>>> print(getlist([1, 2, 3]))
[1, 2, 3, 1, 2, 3]
```

3.6.5 How to Write a Deferrable Function?

The answer is simple: like you would with any other normal function! We've done that already in all the examples we've shown in this documentation. A question that somehow naturally comes up here is whether you can call a deferrable function from within a deferrable function, since this doesn't make a lot of sense: after all, your function will be deferred anyway.

The answer is, yes. You can call other deferrable functions from within a deferrable function. Thanks to the implicit evaluation rules as well as the fact that the return value of a deferrable function is also evaluated if it is a deferred expression, you can write a deferrable function without caring much about whether the functions you call are themselves deferrable or not. However, you should be aware of passing mutable objects to deferrable functions. If these objects

---

happen to change between the actual call and the implicit evaluation of the deferrable function, you might run into surprises. In any case, if you want the immediate evaluation of a deferrable function or expression, you can always do that by calling `evaluate` on it.

The following example demonstrates two different ways writing a deferrable function that checks the average of the elements of an iterable:

```python
import reframe.utility.sanity as sn

@sn.sanity_function
def check_avg_with_deferrables(iterable):
    avg = sn.sum(iterable) / sn.len(iterable)
    return -1 if avg > 2 else 1

@sn.sanity_function
def check_avg_without_deferrables(iterable):
    avg = sum(iterable) / len(iterable)
    return -1 if avg > 2 else 1
```

```python
>>> evaluate(check_avg_with_deferrables([1, 2, 3, 4]))
-1
>>> evaluate(check_avg_without_deferrables([1, 2, 3, 4]))
-1
```

The first version uses the *sum* and *len* functions from *reframe.utility.sanity*, which are deferrable versions of the corresponding builtins. The second version uses directly the builtin `sum` and `len` functions. As you can see, both of them behave in exactly the same way. In the version with the deferrables, `avg` is a deferred expression but it is evaluated by the `if` statement before returning.

Generally, inside a sanity function, it is a preferable to use the non-deferrable version of a function, if that exists, since you avoid the extra overhead and bookkeeping of the deferring mechanism.

3.6.6 Deferrable Sanity Functions

Normally, you will not have to implement your own sanity functions, since ReFrame provides already a variety of them. You can find the complete list of provided sanity functions here.

Similarities and Differences with Generators

Python allows you to create functions that will be evaluated lazily. These are called generator functions. Their key characteristic is that instead of using the `return` keyword to return values, they use the `yield` keyword. I'm not going to go into the details of the generators, since there is plenty of documentation out there, so I will focus on the similarities and differences with our deferrable functions.

Similarities

- Both generators and our deferrables return an object representing the deferred expression when you call them.
- Both generators and deferrables may be evaluated explicitly or implicitly when they appear in certain expressions.
- When you try to iterate over a generator or a deferrable, you trigger its evaluation.

Differences

- You can include deferrables in any arithmetic expression and the result will be another deferrable expression. This is not true with generator functions, which will raise a `TypeError` in such cases or they will always evaluate to `False` if you include them in boolean expressions Here is an example demonstrating this:

```
>>> @sn.sanity_function
... def dsize(iterable):
...     print(len(iterable))
...     return len(iterable)
...
>>> def gsize(iterable):
...     print(len(iterable))
...     yield len(iterable)
...
>>> l = [1, 2]
>>> dsize(l)
<reframe.core.deferrable._DeferredExpression object at 0x2abc630abb38>
>>> gsize(l)
<generator object gsize at 0x2abc62a4bf10>
>>> expr = gsize(l) == 2
>>> expr
False
>>> expr = gsize(l) + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'generator' and 'int'
>>> expr = dsize(l) == 2
>>> expr
<reframe.core.deferrable._DeferredExpression object at 0x2abc630abba8>
>>> expr = dsize(l) + 2
>>> expr
<reframe.core.deferrable._DeferredExpression object at 0x2abc630abc18>
```

Notice that you cannot include generators in expressions, whereas you can generate arbitrary expressions with deferrables.

- Generators are iterator objects, while deferred expressions are not. As a result, you can trigger the evaluation of a generator expression using the `next` builtin function. For a deferred expression you should use `evaluate` instead.

- A generator object is iterable, whereas a deferrable object will be iterable if and only if the result of its evaluation is iterable.

Note: Technically, a deferrable object is iterable, too, since it provides the `__iter__` method. That's why you can include it in iteration expressions. However, it delegates this call to the result of its evaluation.

Here is an example demonstrating this difference:

```
>>> for i in gsize(l): print(i)
...
2
2
>>> for i in dsize(l): print(i)
...
2
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/users/karakasv/Devel/reframe/reframe/core/deferrable.py", line 73, in __
↪iter__
    return iter(self.evaluate())
TypeError: 'int' object is not iterable
```

Notice how the iteration works fine with the generator object, whereas with the deferrable function, the iteration call is delegated to the result of the evaluation, which is not an iterable, therefore yielding `TypeError`. Notice also, the printout of 2 in the iteration over the deferrable expression, which shows that it has been evaluated.

3.7 Running ReFrame

Before getting into any details, the simplest way to invoke ReFrame is the following:

```
./bin/reframe -c /path/to/checks -R --run
```

This will search recursively for test files in `/path/to/checks` and will start running them on the current system.

ReFrame's front-end goes through three phases:

1. Load tests

2. Filter tests

3. Act on tests

In the following, we will elaborate on these phases and the key command-line options controlling them. A detailed listing of all the command-line options grouped by phase is given by `./bin/reframe -h`.

3.7.1 Supported Actions

Even though an action is the last phase that the front-end goes through, we are listing it first since an action is always required. Currently there are only two available actions:

1. Listing of the selected checks

2. Execution of the selected checks

Listing of the regression tests

To retrieve a listing of the selected checks, you must specify the `-l` or `--list` options. An example listing of checks is the following that lists all the tests found under the `tutorial/` folder:

```
./bin/reframe -c tutorial -l
```

The output looks like:

```
Command line: ./bin/reframe -c tutorial/ -l
Reframe version: 2.7
Launched by user: karakasv
Launched on host: daint103
Reframe paths
=============
```

```
    Check prefix      :
    Check search path : 'tutorial/'
    Stage dir prefix  : /users/karakasv/Devel/reframe/stage/
    Output dir prefix : /users/karakasv/Devel/reframe/output/
    Logging dir       : /users/karakasv/Devel/reframe/logs
List of matched checks
======================
  * example1_check (Simple matrix-vector multiplication example)
        tags: [tutorial], maintainers: [you-can-type-your-email-here]
  * example2a_check (Matrix-vector multiplication example with OpenMP)
        tags: [tutorial], maintainers: [you-can-type-your-email-here]
  * example2b_check (Matrix-vector multiplication example with OpenMP)
        tags: [tutorial], maintainers: [you-can-type-your-email-here]
  * example3_check (Matrix-vector multiplication example with MPI)
        tags: [tutorial], maintainers: [you-can-type-your-email-here]
  * example4_check (Matrix-vector multiplication example with OpenACC)
        tags: [tutorial], maintainers: [you-can-type-your-email-here]
  * example5_check (Matrix-vector multiplication example with Cuda)
        tags: [tutorial], maintainers: [you-can-type-your-email-here]
  * example6_check (Matrix-vector multiplication with L2 norm check)
        tags: [tutorial], maintainers: [you-can-type-your-email-here]
  * example7_check (Matrix-vector multiplication example with Cuda)
        tags: [tutorial], maintainers: [you-can-type-your-email-here]
  * example8_serial_check (Serial matrix-vector multiplication)
        tags: [tutorial], maintainers: [you-can-type-your-email-here]
  * example8_openmp_check (OpenMP matrix-vector multiplication)
        tags: [tutorial], maintainers: [you-can-type-your-email-here]
  * example8_mpi_check (MPI matrix-vector multiplication)
        tags: [tutorial], maintainers: [you-can-type-your-email-here]
  * example8_openacc_check (OpenACC matrix-vector multiplication)
        tags: [tutorial], maintainers: [you-can-type-your-email-here]
  * example8_cuda_check (Cuda matrix-vector multiplication)
        tags: [tutorial], maintainers: [you-can-type-your-email-here]
Found 13 check(s).
```

The listing contains the name of the check, its description, the tags associated with it and a list of its maintainers. Note that this listing may also contain checks that are not supported by the current system. These checks will be just skipped if you try to run them.

Execution of the regression tests

To run the regression tests you should specify the *run* action though the `-r` or `--run` options.

Note: The listing action takes precedence over the execution, meaning that if you specify both `-l -r`, only the listing action will be performed.

```
./bin/reframe --notimestamp -c checks/cuda/cuda_checks.py --prefix . -r
```

The output of the regression run looks like the following:

```
Command line: ./bin/reframe -c tutorial/example1.py -r
Reframe version: 2.7
Launched by user: karakasv
```

```
Launched on host: daint103
Reframe paths
=============
    Check prefix      :
    Check search path : 'tutorial/example1.py'
    Stage dir prefix  : /users/karakasv/Devel/reframe/stage/
    Output dir prefix : /users/karakasv/Devel/reframe/output/
    Logging dir       : /users/karakasv/Devel/reframe/logs
[==========] Running 1 check(s)
[==========] Started on Tue Oct 24 18:13:33 2017

[----------] started processing example1_check (Simple matrix-vector multiplication␣
→example)
[ RUN      ] example1_check on daint:mc using PrgEnv-cray
[       OK ] example1_check on daint:mc using PrgEnv-cray
[ RUN      ] example1_check on daint:mc using PrgEnv-gnu
[       OK ] example1_check on daint:mc using PrgEnv-gnu
[ RUN      ] example1_check on daint:mc using PrgEnv-intel
[       OK ] example1_check on daint:mc using PrgEnv-intel
[ RUN      ] example1_check on daint:mc using PrgEnv-pgi
[       OK ] example1_check on daint:mc using PrgEnv-pgi
[ RUN      ] example1_check on daint:gpu using PrgEnv-cray
[       OK ] example1_check on daint:gpu using PrgEnv-cray
[ RUN      ] example1_check on daint:gpu using PrgEnv-gnu
[       OK ] example1_check on daint:gpu using PrgEnv-gnu
[ RUN      ] example1_check on daint:gpu using PrgEnv-intel
[       OK ] example1_check on daint:gpu using PrgEnv-intel
[ RUN      ] example1_check on daint:gpu using PrgEnv-pgi
[       OK ] example1_check on daint:gpu using PrgEnv-pgi
[ RUN      ] example1_check on daint:login using PrgEnv-cray
[       OK ] example1_check on daint:login using PrgEnv-cray
[ RUN      ] example1_check on daint:login using PrgEnv-gnu
[       OK ] example1_check on daint:login using PrgEnv-gnu
[ RUN      ] example1_check on daint:login using PrgEnv-intel
[       OK ] example1_check on daint:login using PrgEnv-intel
[ RUN      ] example1_check on daint:login using PrgEnv-pgi
[       OK ] example1_check on daint:login using PrgEnv-pgi
[----------] finished processing example1_check (Simple matrix-vector multiplication␣
→example)

[  PASSED  ] Ran 12 test case(s) from 1 check(s) (0 failure(s))
[==========] Finished on Tue Oct 24 18:15:06 2017
```

3.7.2 Discovery of Regression Tests

When ReFrame is invoked, it tries to locate regression tests in a predefined path. By default, this path is the
`<reframe-install-dir>/checks`. You can also retrieve this path as follows:

```
./bin/reframe -l | grep 'Check search path'
```

If the path line is prefixed with `(R)`, every directory in that path will be searched recursively for regression tests.

As described extensively in the "ReFrame Tutorial", regression tests in ReFrame are essentially Python source files
that provide a special function, which returns the actual regression test instances. A single source file may also provide
multiple regression tests. ReFrame loads the python source files and tries to call this special function; if this function

---

cannot be found, the source file will be ignored. At the end of this phase, the front-end will have instantiated all the tests found in the path.

You can override the default search path for tests by specifying the `-c` or `--checkpath` options. We have already done that already when listing all the tutorial tests:

```
./bin/reframe -c tutorial/ -l
```

ReFrame the does not search recursively into directories specified with the `-c` option, unless you explicitly specify the `-R` or `--recurse` options.

The `-c` option completely overrides the default path. Currently, there is no option to prepend or append to the default regression path. However, you can build your own check path by specifying multiple times the `-c` option. The `-c`option accepts also regular files. This is very useful when you are implementing new regression tests, since it allows you to run only your test:

```
./bin/reframe -c /path/to/my/new/test.py -r
```

3.7.3 Filtering of Regression Tests

At this phase you can select which regression tests should be run or listed. There are several ways to select regression tests, which we describe in more detail here:

Selecting tests by programming environment

To select tests by the programming environment, use the `-p` or `--prgenv` options:

```
./bin/reframe -p PrgEnv-gnu -l
```

This will select all the checks that support the `PrgEnv-gnu` environment.

You can also specify multiple times the `-p` option, in which case a test will be selected if it support all the programming environments specified in the command line. For example the following will select all the checks that can run with both `PrgEnv-cray` and `PrgEnv-gnu`:

```
./bin/reframe -p PrgEnv-gnu -p PrgEnv-cray -l
```

If you are going to run a set of tests selected by programming environment, they will run only for the selected programming environment(s).

Selecting tests by tags

As we have seen in the "ReFrame tutorial", every regression test may be associated with a set of tags. Using the `-t` or `--tag` option you can select the regression tests associated with a specific tag. For example the following will list all the tests that have a `maintenance` tag:

```
./bin/reframe -t maintenance -l
```

Similarly to the `-p` option, you can chain multiple `-t` options together, in which case a regression test will be selected if it is associated with all the tags specified in the command line. The list of tags associated with a check can be viewed in the listing output when specifying the `-l` option.

Selecting tests by name

It is possible to select or exclude tests by name through the `--name` or `-n` and `--exclude` or `-x` options. For example, you can select only the `example7_check` from the tutorial as follows:

```
./bin/reframe -c tutorial n example7_check -l
```

```
Command line: ./bin/reframe -c tutorial/ -n example7_check -l
Reframe version: 2.7
Launched by user: karakasv
Launched on host: daint103
Reframe paths
=============
    Check prefix      :
    Check search path : 'tutorial/'
    Stage dir prefix  : /users/karakasv/Devel/reframe/stage/
    Output dir prefix : /users/karakasv/Devel/reframe/output/
    Logging dir       : /users/karakasv/Devel/reframe/logs
List of matched checks
======================
  * example7_check (Matrix-vector multiplication example with Cuda)
        tags: [tutorial], maintainers: [you-can-type-your-email-here]
Found 1 check(s).
```

Similarly, you can exclude this test by passing the `-x example7_check` option:

```
Command line: ./bin/reframe -c tutorial/ -x example7_check -l
Reframe version: 2.7
Launched by user: karakasv
Launched on host: daint103
Reframe paths
=============
    Check prefix      :
    Check search path : 'tutorial/'
    Stage dir prefix  : /users/karakasv/Devel/reframe/stage/
    Output dir prefix : /users/karakasv/Devel/reframe/output/
    Logging dir       : /users/karakasv/Devel/reframe/logs
List of matched checks
======================
  * example1_check (Simple matrix-vector multiplication example)
        tags: [tutorial], maintainers: [you-can-type-your-email-here]
  * example2a_check (Matrix-vector multiplication example with OpenMP)
        tags: [tutorial], maintainers: [you-can-type-your-email-here]
  * example2b_check (Matrix-vector multiplication example with OpenMP)
        tags: [tutorial], maintainers: [you-can-type-your-email-here]
  * example3_check (Matrix-vector multiplication example with MPI)
        tags: [tutorial], maintainers: [you-can-type-your-email-here]
  * example4_check (Matrix-vector multiplication example with OpenACC)
        tags: [tutorial], maintainers: [you-can-type-your-email-here]
  * example5_check (Matrix-vector multiplication example with Cuda)
        tags: [tutorial], maintainers: [you-can-type-your-email-here]
  * example6_check (Matrix-vector multiplication with L2 norm check)
        tags: [tutorial], maintainers: [you-can-type-your-email-here]
  * example8_serial_check (Serial matrix-vector multiplication)
        tags: [tutorial], maintainers: [you-can-type-your-email-here]
  * example8_openmp_check (OpenMP matrix-vector multiplication)
        tags: [tutorial], maintainers: [you-can-type-your-email-here]
```

```
   * example8_mpi_check (MPI matrix-vector multiplication)
         tags: [tutorial], maintainers: [you-can-type-your-email-here]
   * example8_openacc_check (OpenACC matrix-vector multiplication)
         tags: [tutorial], maintainers: [you-can-type-your-email-here]
   * example8_cuda_check (Cuda matrix-vector multiplication)
         tags: [tutorial], maintainers: [you-can-type-your-email-here]
Found 12 check(s).
```

3.7.4 Controlling the Execution of Regression Tests

There are several options for controlling the execution of regression tests. Keep in mind that these options will affect all the tests that will run with the current invocation. They are summarized below:

- `-A ACCOUNT`, `--account ACCOUNT`: Submit regression test jobs using `ACCOUNT`.

- `-P PART`, `--partition PART`: Submit regression test jobs in the *scheduler partition* `PART`.

- `--reservation RES`: Submit regression test jobs in reservation `RES`.

- `--nodelist NODELIST`: Run regression test jobs on the nodes specified in `NODELIST`.

- `--exclude-nodes NODELIST`: Do not run the regression test jobs on any of the nodes specified in `NODELIST`.

- `--job-option OPT`: Pass option `OPT` directly to the back-end job scheduler. This option *must* be used with care, since you may break the submission mechanism. All of the above job submission related options could be expressed with this option. For example, the `-n NODELIST` is equivalent to `--job-option='--nodelist=NODELIST'` for a Slurm job scheduler. If you pass an option that is already defined by the framework, the framework will *not* explicitly override it; this is up to scheduler. All extra options defined from the command line are appended to the automatically generated options in the generated batch script file. So if you redefine one of them, e.g., `--output` for the Slurm scheduler, it is up the job scheduler on how to interpret multiple definitions of the same options. In this example, Slurm's policy is that later definitions of options override previous ones. So, in this case, way you would override the standard output for all the submitted jobs!

- `--force-local`: Force the local execution of the selected tests. No jobs will be submitted.

- `--skip-sanity-check`: Skip sanity checking phase.

- `--skip-performance-check`: Skip performance verification phase.

- `--strict`: Force strict performance checking. Some tests may set their `strict_check` attribute to `False` (see "Reference Guide") in order to just let their performance recorded but not yield an error. This option overrides this behavior and forces all tests to be strict.

- `--skip-system-check`: Skips the system check and run the selected tests even if they do not support the current system. This option is sometimes useful when you need to quickly verify if a regression test supports a new system.

- `--skip-prgenv-check`: Skips programming environment check and run the selected tests for even if they do not support a programming environment. This option is useful when you need to quickly verify if a regression check supports another programming environment. For example, if you know that a tests supports only `PrgEnv-cray` and you need to check if it also works with `PrgEnv-gnu`, you can test is as follows:

```
./bin/reframe -c /path/to/my/check.py -p PrgEnv-gnu --skip-prgenv-check -r
```

3.7.5 Configuring ReFrame Directories

ReFrame uses three basic directories during the execution of tests:

1. The stage directory

 • Each regression test is executed in a "sandbox"; all of its resources (source files, resources) are copied over to a stage directory (if the directory preexists, it will be wiped out) and executed from there. This will also be the working directory for the test.

2. The output directory

 • After a regression test finishes some important files will be copied from the stage directory to the output directory (if the directory preexists, it will be wiped out). By default these are the standard output, standard error and the generated job script file. A regression test may also specify to keep additional files.

3. The log directory

 • This is where the performance log files of the individual performance tests are placed (see *Logging* for more information)

By default, all these directories are placed under a common prefix, which defaults to `..` The rest of the directories are organized as follows:

 • Stage directory: `${prefix}/stage/<timestamp>`

 • Output directory: `${prefix}/output/<timestamp>`

 • Performance log directory: `${prefix}/logs`

You can optionally append a timestamp directory component to the above paths (except the logs directory), by using the `--timestamp` option. This options takes an optional argument to specify the timestamp format. The default time format is `%FT%T`, which results into timestamps of the form `2017-10-24T21:10:29`.

You can override either the default global prefix or any of the default individual directories using the corresponding options.

 • `--prefix DIR`: set prefix to `DIR`.

 • `--output DIR`: set output directory to `DIR`.

 • `--stage DIR`: set stage directory to `DIR`.

 • `--logdir DIR`: set performance log directory to `DIR`.

The stage and output directories are created only when you run a regression test. However you can view the directories that will be created even when you do a listing of the available checks with the `-l` option. This is useful if you want to check the directories that ReFrame will create.

```
./bin/reframe --prefix /foo -l
```

```
Command line: ./bin/reframe --prefix /foo -t foo -l
Reframe version: 2.7
Launched by user: karakasv
Launched on host: daint103
Reframe paths
=============
    Check prefix      : /users/karakasv/Devel/reframe
(R) Check search path : 'checks/'
    Stage dir prefix  : /foo/stage/
    Output dir prefix : /foo/output/
    Logging dir       : /foo/logs
```

(continues on next page)

```
List of matched checks
======================
Found 0 check(s).
```

You can also define different default directories per system by specifying them in the site configuration settings file. The command line options, though, take always precedence over any default directory.

3.7.6 Logging

From version 2.4 onward, ReFrame supports logging of its actions. ReFrame creates two files inside the current working directory every time it is run:

- `reframe.out`: This file stores the output of a run as it was printed in the standard output.

- `reframe.log`: This file stores more detailed of information on ReFrame's actions.

By default, the output in `reframe.log` looks like the following:

```
[2017-10-24T21:19:04] info: reframe: [----------] started processing example7_check␣
↪(Matrix-vector mult
iplication example with Cuda)
[2017-10-24T21:19:04] info: reframe: [   SKIP   ] skipping daint:mc
[2017-10-24T21:19:04] info: reframe: [ RUN      ] example7_check on daint:gpu using␣
↪PrgEnv-cray
[2017-10-24T21:19:04] debug: example7_check: setting up the environment
[2017-10-24T21:19:04] debug: example7_check: loading environment for partition␣
↪daint:gpu
[2017-10-24T21:19:05] debug: example7_check: loading environment PrgEnv-cray
[2017-10-24T21:19:05] debug: example7_check: setting up paths
[2017-10-24T21:19:05] debug: example7_check: setting up the job descriptor
[2017-10-24T21:19:05] debug: example7_check: job scheduler backend: nativeslurm
[2017-10-24T21:19:05] debug: example7_check: setting up performance logging
[2017-10-24T21:19:05] debug: example7_check: compilation started
[2017-10-24T21:19:06] debug: example7_check: compilation stdout:

[2017-10-24T21:19:06] debug: example7_check: compilation stderr:
nvcc warning : The 'compute_20', 'sm_20', and 'sm_21' architectures are deprecated,␣
↪and may be removed
in a future release (Use -Wno-deprecated-gpu-targets to suppress warning).

[2017-10-24T21:19:06] debug: example7_check: compilation finished
[2017-10-24T21:19:09] debug: example7_check: spawned job (jobid=4163846)
[2017-10-24T21:19:21] debug: example7_check: spawned job finished
[2017-10-24T21:19:21] debug: example7_check: copying interesting files to output␣
↪directory
[2017-10-24T21:19:21] debug: example7_check: removing stage directory
[2017-10-24T21:19:21] info: reframe: [       OK ] example7_check on daint:gpu using␣
↪PrgEnv-cray
```

Each line starts with a timestamp, the level of the message (`info`, `debug` etc.), the context in which the framework is currently executing (either `reframe` or the name of the current test and, finally, the actual message.

Every time ReFrame is run, both `reframe.out` and `reframe.log` files will be rewritten. However, you can ask ReFrame to copy them to the output directory before exiting by passing it the `--save-log-files` option.

Configuring logging

You can configure several aspects of logging in ReFrame and even how the output will look like. ReFrame's logging mechanism is built upon Python's logging framework adding extra logging levels and more formatting capabilities.

Logging in ReFrame is configured by the `_logging_config` variable in the `reframe/settings.py` file. The default configuration looks as follows:

```python
_logging_config = {
    'level': 'DEBUG',
    'handlers': {
        'reframe.log' : {
            'level'    : 'DEBUG',
            'format'   : '[%(asctime)s] %(levelname)s: '
                         '%(testcase_name)s: %(message)s',
            'append'   : False,
        },

        # Output handling
        '&1': {
            'level'    : 'INFO',
            'format'   : '%(message)s'
        },
        'reframe.out' : {
            'level'    : 'INFO',
            'format'   : '%(message)s',
            'append'   : False,
        }
    }
}
```

Note that this configuration dictionary is not the same as the one used by Python's logging framework. It is a simplified version adapted to the needs of ReFrame.

The `_logging_config` dictionary has two main key entries:

- `level` (default: `'INFO'`): This is the lowest level of messages that will be passed down to the different log record handlers. Any message with a lower level than that, it will be filtered out immediately and will not be passed to any handler. ReFrame defines the following logging levels with a decreasing severity: `CRITICAL`, `ERROR`, `WARNING`, `INFO`, `VERBOSE` and `DEBUG`. Note that the level name is *not* case sensitive in ReFrame.

- `handlers`: A dictionary defining the properties of the handlers that are attached to ReFrame's logging mechanism. The key is either a filename or a special character combination denoting standard output (`&1`) or standard error (`&2`). You can attach as many handlers as you like. The value of each handler key is another dictionary that holds the properties of the corresponding handler as key/value pairs.

The configurable properties of a log record handler are the following:

- `level` (default: `'debug'`): The lowest level of log records that this handler can process.

- `format` (default: `'%(message)s'`): Format string for the printout of the log record. ReFrame supports all the format strings from Python's logging library and provides the following additional ones:

  - `check_name`: Prints the name of the regression test on behalf of which ReFrame is currently executing. If ReFrame is not in the context of regression test, `reframe` will be printed.

  - `check_jobid`: Prints the job or process id of the job or process associated with currently executing regression test. If a job or process is not yet created, `-1` will be printed.

  - `testcase_name`: Print the name of the test case that is currently executing. Test case is essentially a tuple consisting of the test name, the current system and partition and the current programming environment.

This format string prints out like `<test-name>@<partition> using <environ>`.

- `datefmt` (default: `'%FT%T'`) The format that will be used for outputting timestamps (i.e., the `%(asctime)s` field). Acceptable formats must conform to standard library's time.strftime() function.

- `append` (default: `False`) Controls whether ReFrame should append to this file or not. This is ignored for the standard output/error handlers.

- `timestamp` (default: `None`): Append a timestamp to this log filename. This property may accept any date format as the `datefmt` property. If set for a `filename.log` handler entry, the resulting log file name will be `filename_<timestamp>.log`. This property is ignored for the standard output/error handlers.

Performance Logging

ReFrame supports additional logging for performance tests specifically, in order to record historical performance data. For each performance test, a log file of the form `<test-name>.log` is created under the ReFrame's *log directory* where the test's performance is recorded. The default format used for this file is `'[%(asctime)s] %(testcase_name)s (jobid=%(check_jobid)s):  %(message)s'` and ReFrame always appends to this file. Currently, it is not possible for users to configure performance logging.

The resulting log file looks like the following:

```
[2017-12-01T15:31:20] example7_check@daint:gpu using PrgEnv-cray (jobid=649790):␣
→value: 47.797996, reference: (50.0, -0.1, 0.1)
[2017-12-01T15:31:24] example7_check@daint:gpu using PrgEnv-gnu (jobid=649791):␣
→value: 49.048228, reference: (50.0, -0.1, 0.1)
[2017-12-01T15:31:24] example7_check@daint:gpu using PrgEnv-pgi (jobid=649792):␣
→value: 48.575334, reference: (50.0, -0.1, 0.1)
```

The interpretation of the performance values depends on the individual tests. The above output is from the CUDA performance test we presented in the tutorial, so the value refers to the achieved Gflop/s. The reference value is a three-element tuple of the form (`<reference>`, `<lower-threshold>`, `<upper-threshold>`), where the `lower-threshold` and `upper-threshold` are the acceptable tolerance thresholds expressed in percentages. For example, the performance check shown above has a reference value of 50 Gflop/s ± 10%.

3.7.7 Asynchronous Execution of Regression Checks

From version 2.4, ReFrame supports asynchronous execution of regression tests. This execution policy can be enabled by passing the option `--exec-policy=async` to the command line. The default execution policy is `serial` which enforces a sequential execution of the selected regression tests. The asynchronous execution policy parallelizes only the running phase of the tests. The rest of the phases remain sequential.

A limit of concurrent jobs (pending and running) may be configured for each virtual system partition. As soon as the concurrency limit of a partition is reached, ReFrame will hold the execution of new regression tests until a slot is released in that partition.

When executing in asynchronous mode, ReFrame's output differs from the sequential execution. The final result of the tests will be printed at the end and additional messages may be printed to indicate that a test is held. Here is an example output of ReFrame using asynchronous execution policy:

```
Command line: ./reframe.py -c tutorial/ --exec-policy=async -r
Reframe version: 2.7
Launched by user: karakasv
Launched on host: daint104
Reframe paths
=============
```

```
    Check prefix       :
    Check search path : 'tutorial/'
    Stage dir prefix  : /users/karakasv/Devel/reframe/stage/
    Output dir prefix : /users/karakasv/Devel/reframe/output/
    Logging dir       : /users/karakasv/Devel/reframe/logs
[==========] Running 13 check(s)
[==========] Started on Sun Nov  5 19:37:09 2017


[----------] started processing example1_check (Simple matrix-vector multiplication␣
↪example)
[ RUN      ] example1_check on daint:login using PrgEnv-cray
[ RUN      ] example1_check on daint:login using PrgEnv-gnu
[ RUN      ] example1_check on daint:login using PrgEnv-intel
[ RUN      ] example1_check on daint:login using PrgEnv-pgi
[ RUN      ] example1_check on daint:gpu using PrgEnv-cray
[ RUN      ] example1_check on daint:gpu using PrgEnv-gnu
[ RUN      ] example1_check on daint:gpu using PrgEnv-intel
[ RUN      ] example1_check on daint:gpu using PrgEnv-pgi
[ RUN      ] example1_check on daint:mc using PrgEnv-cray
[ RUN      ] example1_check on daint:mc using PrgEnv-gnu
[ RUN      ] example1_check on daint:mc using PrgEnv-intel
[ RUN      ] example1_check on daint:mc using PrgEnv-pgi
[----------] finished processing example1_check (Simple matrix-vector multiplication␣
↪example)


...

[----------] started processing example8_cuda_check (Cuda matrix-vector␣
↪multiplication)
[   SKIP   ] skipping daint:login
[ RUN      ] example8_cuda_check on daint:gpu using PrgEnv-cray
[ RUN      ] example8_cuda_check on daint:gpu using PrgEnv-gnu
[   SKIP   ] skipping PrgEnv-intel for daint:gpu
[ RUN      ] example8_cuda_check on daint:gpu using PrgEnv-pgi
[   SKIP   ] skipping daint:mc
[----------] finished processing example8_cuda_check (Cuda matrix-vector␣
↪multiplication)

[----------] waiting for spawned checks
[       OK ] example1_check on daint:login using PrgEnv-cray
[       OK ] example1_check on daint:login using PrgEnv-gnu
[       OK ] example1_check on daint:login using PrgEnv-intel
[       OK ] example1_check on daint:login using PrgEnv-pgi
[       OK ] example1_check on daint:gpu using PrgEnv-cray
[       OK ] example1_check on daint:gpu using PrgEnv-gnu
[       OK ] example1_check on daint:gpu using PrgEnv-intel
[       OK ] example1_check on daint:gpu using PrgEnv-pgi
[       OK ] example1_check on daint:mc using PrgEnv-cray
[       OK ] example1_check on daint:mc using PrgEnv-gnu
[       OK ] example1_check on daint:mc using PrgEnv-intel
[       OK ] example1_check on daint:mc using PrgEnv-pgi
...
[       OK ] example8_openacc_check on daint:gpu using PrgEnv-cray
[       OK ] example8_openacc_check on daint:gpu using PrgEnv-pgi
[       OK ] example8_cuda_check on daint:gpu using PrgEnv-cray
[       OK ] example8_cuda_check on daint:gpu using PrgEnv-gnu
[       OK ] example8_cuda_check on daint:gpu using PrgEnv-pgi
```

```
[----------] all spawned checks finished
[  PASSED  ] Ran 97 test case(s) from 13 check(s) (0 failure(s))
[==========] Finished on Sun Nov  5 19:42:23 2017
```

The asynchronous execution policy may provide significant overall performance benefits for run-only regression tests. For compile-only and normal tests that require a compilation, the execution time will be bound by the total compilation time of the test.

3.8 Use Cases

3.8.1 ReFrame Usage at CSCS

The ReFrame framework has been in production at CSCS since December 2016. We use it to test not only Piz Daint, but almost all our systems that we provide to users.

We have two large sets of regression tests:

- production tests and

- maintenance tests.

Tags are used to mark these categories and a regression test may belong to both of them. Production tests are run daily to monitor the sanity of the system and its performance. All performance tests log their performance values. The performance over time of certain applications are monitored graphically using Grafana.

The total set of our regression tests comprises 172 individual tests, from which 153 are marked as production tests. Some of them are eligible to run on both the multicore and hybrid partitions of the system, whereas others are meant to run only on the login nodes. Depending on the test, multiple programming environments might be tried. In total, 448 test cases are run from the 153 regression tests on all the system partitions. The following Table summarizes the production regression tests.

The set of maintenance regression tests is much more limited to decrease the downtime of the system. The regression suite runs at the beginning of the maintenance session and just before returning the machine to the users, so that we can ensure that the user experience is at least at the level before the system was taken down. The maintenance set of tests comprises application performance tests, some GPU library performance checks, Slurm checks and some POSIX filesystem checks.

The porting of the regression suite to the MeteoSwiss production system Piz Kesch, using ReFrame was almost trivial. The new system entry was added in the framework's configuration file describing the different partitions together with a new redefined `PrgEnv-gnu` environment to use different compiler wrappers. Porting the regression tests of interest was also a straightforward process. In most of the cases, adding just the corresponding system partitions to the `valid_systems` variables and adjusting accordingly the `valid_prog_environs` was enough.

ReFrame really focuses on abstracting away all the gory details from the regression test description, hence letting the user to concentrate solely on the logic of his test. A bit of this effect can be seen in the following Table where the total amount of lines of code (loc) of the regression tests written in the previous shell script-based solution and ReFrame is shown. We also present a snapshot of the first public release of ReFrame (v2.2).

Maintenance Burden	Shell-Script Based	ReFrame (May 2017)	ReFrame (Nov 2017)
Total tests	179	122	172
Total size of tests	14635 loc	2985 loc	4493 loc
Avg. test file size	179 loc	93 loc	87 loc
Avg. effective test size	179 loc	25 loc	25 loc

The difference in the total amount of regression test code is dramatic. From the 15K lines of code of the old shell script based regression testing suite, ReFrame tests use only 3K lines of code (first release) achieving a higher coverage.

Note: The higher test count of the older suite refers to test cases, i.e., running the same test for different programming environments, whereas for ReFrame the counts do not account for this.

Each regression test file in ReFrame is 80–90 loc on average. However, each regression test file may contain or generate more than one related tests, thus leading to the effective decrease of the line count per test to only 25 loc.

Separating the logical description of a regression test from all the unnecessary implementation details contributes significantly in the ease of writing and maintaining new regression tests with ReFrame.

3.9 About ReFrame

3.9.1 What Is ReFrame?

ReFrame is a framework developed by CSCS to facilitate the writing of regression tests that check the sanity of HPC systems. Its main goal is to allow users to write their own regression tests without having to deal with all the details of setting up the environment for the test, querying the status of their job, managing the output of the job and looking for sanity and/or performance results. Users should be concerned only about the logical requirements of their tests. This allows users' regression checks to be maintained and adapted to new systems easily.

The user describes his test in a simple Python class and the framework takes care of all the details of the low-level interaction with the system. The framework is structured in such a way that with a basic knowledge of Python and minimal coding a user can write a regression test, which will be able to run out-of-the-box on a variety of systems and programming environments.

Writing regression tests in a high-level language, such as Python, allows users to take advantage of the language's higher expressiveness and bigger capabilities compared to classical shell scripting, which is the norm in HPC testing. This could lead to a more manageable code base of regression tests with significantly reduced maintenance costs.

3.9.2 ReFrame's Goals

When designing the framework we have set three major goals:

Productivity The writer of a regression test should focus only on the logical structure and requirements of the test and should not need to deal with any of the low level details of interacting with the system, e.g., how the environment of the test is loaded, how the associated job is created and has its status checked, how the output parsing is performed etc.

Portability Configuring the framework to support new systems and system configurations should be easy and should not affect the existing tests. Also, adding support of a new system in a regression test should require minimal adjustments.

Robustness and ease of use The new framework must be stable enough and easy to use by non-advanced users. When the system needs to be returned to users outside normal working hours the personnel in charge should be able to run the regression suite and verify the sanity of the system with a minimal involvement.

3.9.3 Why ReFrame?

HPC systems are highly complex systems in all levels of integration; from the physical infrastructure up to the software stack provided to the users. A small change in any of these levels could have an impact on the stability or the

performance of the system perceived by the end users. It is of crucial importance, therefore, not only to make sure that the system is in a sane condition after every maintenance before handing it off to users, but also to monitor its performance during production, so that possible problems are detected early enough and the quality of service is not compromised.

Regression testing can provide a reliable way to ensure the stability and the performance requirements of the system, provided that sufficient tests exist that cover a wide aspect of the system's operations from both the operators' and users' point of view. However, given the complexity of HPC systems, writing and maintaining regression tests can be a very time consuming task. A small change in system configuration or deployment may require adapting hundreds of regression tests at the same time. Similarly, porting a test to a different system may require significant effort if the new system's configuration is substantially different than that of the system that it was originally written for.

ReFrame was designed to help HPC support teams to easily write tests that

- monitor the impact of changes to the system that would affect negatively the users,

- monitor system performance,

- monitor system stability and

- guarantee quality of service.

And also decrease the amount of time and resources required to

- write and maintain regression tests and

- port regression tests to other HPC systems.

3.10 Reference Guide

class reframe.core.pipeline.**CompileOnlyRegressionTest**(**args*, ***kwargs*)
    Bases: *reframe.core.pipeline.RegressionTest*

    Base class for compile-only regression tests.

    These tests are by default local and will skip the run phase of the regression test pipeline.

    The standard output and standard error of the test will be set to those of the compilation stage.

    compile(***compile_opts*)
        The compilation stage of the regression test pipeline.

        The standard output and standard error of this stage will be used as the standard output and error of the test.

    run()
        The run stage of the regression test pipeline.

        Implemented as no-op.

    setup(*partition*, *environ*, ***job_opts*)
        The setup stage of the regression test pipeline.

        Similar to the *RegressionTest.setup()*, except that no job descriptor is set up for this test.

    wait()
        Wait for this test to finish.

        Implemented as no-op

class reframe.core.pipeline.**RegressionTest**(*name*, *prefix*, *system*, *resources*)
    Bases: object

Base class for regression tests.

All regression tests must eventually inherit from this class. This class provides the implementation of the pipeline phases that the regression test goes through during its lifetime.

> **Parameters**
>
> - **name** – The name of the test. This is the only argument that the users may specify freely.
>
> - **prefix** – The directory prefix of the test. You should initialize this to the directory containing the file that defines the regression test. You can achieve this by always passing `os.path.dirname(__file__)`.
>
> - **system** – The system that this regression test will run on. The framework takes care of initializing and passing correctly this argument.
>
> - **resources** – An object managing the framework's resources. The framework takes care of initializing and passing correctly this argument.

Concrete regression test subclasses should call the base constructor as follows:

```python
class MyTest(RegressionTest):
    def __init__(self, my_test_args, **kwargs):
        super().__init__('mytest', os.path.dirname(__file__), **kwargs)
```

check_performance()
> The performance checking phase of the regression test pipeline.
>
> > **Raises** `reframe.core.exceptions.SanityError` – If the performance check fails.

check_sanity()
> The sanity checking phase of the regression test pipeline.
>
> > **Raises** `reframe.core.exceptions.SanityError` – If the sanity check fails.

cleanup(*remove_files=False*, *unload_env=True*)
> The cleanup phase of the regression test pipeline.
>
> > **Parameters**
> >
> > - **remove_files** – If `True`, the stage directory associated with this test will be removed.
> >
> > - **unload_env** – If `True`, the environment that was used to run this test will be unloaded.

compile(***compile_opts*)
> The compilation phase of the regression test pipeline.
>
> > **Parameters** **compile_opts** – Extra options to be passed to the programming environment for compiling the source code of the test.
> >
> > **Raises** `reframe.core.exceptions.ReframeError` – In case of errors.

current_environ
> The programming environment that the regression test is currently executing with.
>
> This is set by the framework during the *setup()* phase.
>
> > **Type** *reframe.core.environments.Environment*.

current_partition
> The system partition the regression test is currently executing on.
>
> This is set by the framework during the *setup()* phase.
>
> > **Type** *reframe.core.systems.SystemPartition*.

current_system
    The system the regression test is currently executing on.

    This is set by the framework during the initialization phase.

        Type *reframe.core.systems.System*.

descr
    A detailed description of the test.

        Type str

        Default self.name

exclusive_access
    Specify whether this test needs exclusive access to nodes.

        Type boolean

        Default False

executable
    The name of the executable to be launched during the run phase.

        Type str

        Default os.path.join('.', self.name)

executable_opts
    List of options to be passed to the *executable*.

        Type list[str]

        Default []

extra_resources
    Extra resources for this test.

    This field is for specifying custom resources needed by this test. These resources are defined in the *configuration* of a system partition. For example, assume that two additional resources, named gpu and datawarp, are defined in the configuration file as follows:

```
'resources': {
    'gpu': [
        '--gres=gpu:{num_gpus_per_node}'
    ],
    'datawarp': [
        '#DW jobdw capacity={capacity}',
        '#DW stage_in source={stagein_src}'
    ]
}
```

    A regression test then may instantiate the above resources by setting the *extra_resources* attribute as follows:

```
self.extra_resources = {
    'gpu': {'num_gpus_per_node': 2}
    'datawarp': {
        'capacity': '100GB',
        'stagein_src': '/foo'
    }
}
```

    The generated batch script (for Slurm) will then contain the following lines:

```
#SBATCH --gres=gpu:2
#DW jobdw capacity=100GB
#DW stage_in source=/foo
```

Notice that if the resource specified in the configuration uses an alternative directive prefix (in this case #DW), this will replace the standard prefix of the backend scheduler (in this case #SBATCH)

If the resource name specified in this variable does not match a resource name in the partition configuration, it will be simply ignored. The *num_gpus_per_node* attribute translates internally to the _rfm_gpu resource, so that setting self.num_gpus_per_node = 2 is equivalent to the following:

```
self.extra_resources = {'_rfm_gpu': {'num_gpus_per_node': 2}}
```

> **Type** dict[str, dict[str, object]]
>
> **Default** {}

Note: New in version 2.8.

Changed in version 2.9.

A new more powerful syntax was introduced that allows also custom job script directive prefixes.

---

is_local()
> Check if the test will execute locally.
>
> A test executes locally if the *local* attribute is set or if the current partition's scheduler does not support job submission.

job
> The job descriptor associated with this test.
>
> This is set by the framework during the *setup()* phase.
>
> > **Type** *reframe.core.schedulers.Job*.

keep_files
> List of files to be kept after the test finishes.
>
> By default, the framework saves the standard output, the standard error and the generated shell script that was used to run this test.
>
> These files will be copied over to the framework's output directory during the *cleanup()* phase.
>
> Directories are also accepted in this field.
>
> Relative path names are resolved against the stage directory.
>
> > **Type** list[str]
> >
> > **Default** []

local
> Always execute this test locally.
>
> > **Type** boolean
> >
> > **Default** False

logger
> A logger associated with the this test.

You can use this logger to log information for your test.

maintainers
> List of people responsible for this test.
>
> When the test fails, this contact list will be printed out.
>
> > **Type** `list[str]`
> >
> > **Default** `[]`

modules
> List of modules to be loaded before running this test.
>
> These modules will be loaded during the *setup()* phase.
>
> > **Type** `list[str]`
> >
> > **Default** `[]`

name
> The name of the test.
>
> > **Type** Alphanumeric string.

num_cpus_per_task
> Number of CPUs per task required by this test.
>
> Ignored if `None`.
>
> > **Type** integral or `None`
> >
> > **Default** `None`

num_gpus_per_node
> Number of GPUs per node required by this test.
>
> > **Type** integral
> >
> > **Default** `0`

num_tasks
> Number of tasks required by this test.
>
> If the number of tasks is set to `0`, ReFrame will try to use all the available nodes of a reservation. A reservation *must* be specified through the *–reservation* command-line option, otherwise the regression test will fail during submission. ReFrame will try to run the test on all the nodes of the reservation that satisfy the selection criteria of the current virtual partition (i.e., constraints and/or partitions).
>
> > **Type** integral
> >
> > **Default** `1`

> **Note:** Changed in version 2.9: Added support for running the test using all the nodes of the specified reservation if the number of tasks is set to `0`.

---

num_tasks_per_core
> Number of tasks per core required by this test.
>
> Ignored if `None`.
>
> > **Type** integral or `None`
> >
> > **Default** `None`

num_tasks_per_node

Number of tasks per node required by this test.

Ignored if `None`.

> **Type** integral or `None`

> **Default** `None`

num_tasks_per_socket

Number of tasks per socket required by this test.

Ignored if `None`.

> **Type** integral or `None`

> **Default** `None`

perf_patterns

Patterns for verifying the performance of this test.

Refer to the *ReFrame Tutorial* for concrete usage examples.

If set to `None`, no performance checking will be performed.

> **Type** A dictionary with keys of type `str` and deferrable expressions (i.e., the result of a *sanity function*) as values. `None` is also allowed.

> **Default** `None`

poll()

Poll the test's state.

> **Returns**

> > `True` if the associated job has finished, `False` otherwise.

> > If no job descriptor is yet associated with this test, `True` is returned.

> **Raises** `reframe.core.exceptions.ReframeError` – In case of errors.

postbuild_cmd

List of shell commands to be executed after a successful compilation.

These commands are executed during the compilation phase and from inside the stage directory. **Each entry in the list spawns a new shell.**

> **Type** `list[str]`

> **Default** `[]`

prebuild_cmd

List of shell commands to be executed before compiling.

These commands are executed during the compilation phase and from inside the stage directory. **Each entry in the list spawns a new shell.**

> **Type** `list[str]`

> **Default** `[]`

prefix

The prefix directory of the test.

> **Type** `str`.

readonly_files
> List of files or directories (relative to the *[sourcesdir](#)*) that will be symlinked in the stage directory and not copied.
>
> You can use this variable to avoid copying very large files to the stage directory.
>
> > **Type** `list[str]`
> >
> > **Default** `[]`

reference
> The set of reference values for this test.
>
> Refer to the *[ReFrame Tutorial](#)* for concrete usage examples.
>
> > **Type** A scoped dictionary with system names as scopes or `None`
> >
> > **Default** `{}`

run()
> The run phase of the regression test pipeline.
>
> This call is non-blocking. It simply submits the job associated with this test and returns.

sanity_patterns
> Refer to the *[ReFrame Tutorial](#)* for concrete usage examples.
>
> If set to `None`, a sanity error will be raised during sanity checking.
>
> > **Type** A deferrable expression (i.e., the result of a *[sanity function](#)*) or `None`
> >
> > **Default** `None`

> **Note:** Changed in version 2.9: The default behaviour has changed and it is now considered a sanity failure if this attribute is set to `None`.
>
> If a test doesn't care about its output, this must be stated explicitly as follows:
>
> ```
> self.sanity_patterns = sn.assert_found(r'.*', self.stdout)
> ```

---

setup(*partition*, *environ*, ***job_opts*)
> The setup phase of the regression test pipeline.
>
> > **Parameters**
> >
> > - **partition** – The system partition to set up this test for.
> >
> > - **environ** – The environment to set up this test for.
> >
> > - **job_opts** – Options to be passed through to the backend scheduler. When overriding this method users should always pass through `job_opts` to the base class method.
> >
> > **Raises** **reframe.core.exceptions.ReframeError** – In case of errors.

sourcepath
> The path to the source file or source directory of the test.
>
> If not absolute, it is resolved against the *[sourcesdir](#)* directory.
>
> If it refers to a regular file, this file will be compiled (its language will be automatically recognized) and the produced executable will be placed in the test's stage directory. If it refers to a directory, this will be copied to the test's stage directory and `make` will be invoked in that.
>
> > **Type** `str`

> **Default** `''`

sourcesdir

> The directory containing the test's resources.
>
> If set to `None`, the test has no resources.
>
> > **Type** `str` or `None`
> >
> > **Default** `os.path.join(self.prefix, 'src')`

stagedir

> The stage directory of the test.
>
> This is set during the *`setup()`* phase.
>
> > **Type** `str`.

stderr

> The name of the file containing the standard error of the test.
>
> This is set during the *`setup()`* phase.
>
> This attribute is evaluated lazily, so it can by used inside sanity expressions.
>
> > **Type** `str`.

stdout

> The name of the file containing the standard output of the test.
>
> This is set during the *`setup()`* phase.
>
> This attribute is evaluated lazily, so it can by used inside sanity expressions.
>
> > **Type** `str`.

strict_check

> Mark this test as a strict performance test.
>
> If a test is marked as non-strict, the performance checking phase will always succeed, unless the `--strict` command-line option is passed when invoking ReFrame.
>
> > **Type** boolean
> >
> > **Default** `True`

tags

> Set of tags associated with this test.
>
> This test can be selected from the frontend using any of these tags.
>
> > **Type** `set[str]`
> >
> > **Default** an empty set

time_limit

> Time limit for this test.
>
> Time limit is specified as a three-tuple in the form `(hh, mm, ss)`, with `hh >= 0, 0 <= mm <= 59` and `0 <= ss <= 59`.
>
> > **Type** `tuple[int]`
> >
> > **Default** `(0, 10, 0)`

use_multithreading

> Specify whether this tests needs simultaneous multithreading enabled.

Ignored if `None`.

> **Type** boolean or `None`
>
> **Default** `None`

valid_prog_environs
    List of programming environmets supported by this test.

> **Type** `list[str]`
>
> **Default** `[]`

valid_systems
    List of systems supported by this test. The general syntax for systems is `<sysname>[:<partname]`.

> **Type** `list[str]`
>
> **Default** `[]`

variables
    Environment variables to be set before running this test.

    These variables will be set during the *setup()* phase.

> **Type** `dict[str, str]`
>
> **Default** `{}`

wait()
    Wait for this test to finish.

> **Raises** **reframe.core.exceptions.ReframeError** – In case of errors.

class reframe.core.pipeline.**RunOnlyRegressionTest**(*name*, *prefix*, *system*, *resources*)
    Bases: *reframe.core.pipeline.RegressionTest*

    Base class for run-only regression tests.

compile(***compile_opts*)
    The compilation phase of the regression test pipeline.

    This is a no-op for this type of test.

run()
    The run phase of the regression test pipeline.

    The resources of the test are copied to the stage directory and the rest of execution is delegated to the *RegressionTest.run()*.

class reframe.core.environments.**Environment**(*name*, *modules=[]*, *variables={}*, ***kwargs*)
    Bases: `object`

    This class abstracts away an environment to run regression tests.

    It is simply a collection of modules to be loaded and environment variables to be set when this environment is loaded by the framework. Users may not create or modify directly environments.

is_loaded
    `True` if this environment is loaded, `False` otherwise.

modules
    The modules associated with this environment.

> **Type** `list` of `str`

name
> The name of this environment.
>
> > **Type** `str`

variables
> The environment variables associated with this environment.
>
> > **Type** dictionary of `str` keys/values.

class reframe.core.environments.**EnvironmentSnapshot**(*name='env_snapshot'*)
> Bases: *reframe.core.environments.Environment*

class reframe.core.environments.**ProgEnvironment**(*name*, *modules=[]*, *variables={}*, *cc='cc'*, *cxx='CC'*, *ftn='ftn'*, *cppflags=None*, *cflags=None*, *cxxflags=None*, *fflags=None*, *ld-flags=None*, ***kwargs*)
> Bases: *reframe.core.environments.Environment*

> A class representing a programming environment.

> This type of environment adds also attributes for setting the compiler and compilation flags.

> If compilation flags are set to `None` (the default, if not set otherwise in ReFrame's configuration), they are not passed to the `make` invocation.

> If you want to disable completely the propagation of the compilation flags to the `make` invocation, even if they are set, you should set the *propagate* attribute to `False`.

cc
> The C compiler of this programming environment.
>
> > **Type** `str`

cflags
> The C compiler flags of this programming environment.
>
> > **Type** `str` or `None`

cppflags
> The preprocessor flags of this programming environment.
>
> > **Type** `str` or `None`

cxx
> The C++ compiler of this programming environment.
>
> > **Type** `str` or `None`

cxxflags
> The C++ compiler flags of this programming environment.
>
> > **Type** `str` or `None`

fflags
> The Fortran compiler flags of this programming environment.
>
> > **Type** `str` or `None`

ftn
> The Fortran compiler of this programming environment.
>
> > **Type** `str` or `None`

include_search_path
    The include search path of this programming environment.

        Type `list` of `str`

        Default `[]`

ldflags
    The linker flags of this programming environment.

        Type `str` or None

propagate
    Propagate the compilation flags to the `make` invocation.

        Type `bool`

        Default True

class reframe.core.environments.**save_environment**
    Bases: `object`

    A context manager for saving and restoring the current environment.

class reframe.core.systems.**System**(*name*, *descr=None*, *hostnames=[]*, *partitions=[]*, *prefix='.'*,
                                           stagedir=None, *outputdir=None*, *logdir=None*, *resources-*
                                           dir='.', *modules_system=None*)
    Bases: `object`

    A representation of a system inside ReFrame.

descr
    The description of this system.

name
    The name of this system.

partition(*name*)
    Get system partition with `name`.

        Returns the requested *SystemPartition*, or None if not found.

partitions
    Get all the active partitions of this system.

        Returns a list of *SystemPartition*.

resourcesdir
    Global resources directory for this system

    You may use this directory for storing large resource files of your regression tests. See here on how to
    configure this.

        Type `str`

class reframe.core.systems.**SystemPartition**(*name*,   *descr=None*,   *scheduler=None*,
                                                    launcher=None,   *access=[]*,   *environs=[]*,
                                                    resources={}, *local_env=None*, *max_jobs=1*)
    Bases: `object`

    A representation of a system partition inside ReFrame.

descr
    A detailed description of this partition.

fullname
> Return the fully-qualified name of this partition.
>
> The fully-qualified name is of the form `<parent-system-name>:<partition-name>`.
>
> > **Type** *str*

launcher
> The type of the backend launcher of this partition.
>
> > **Returns** a subclass of *reframe.core.launchers.JobLauncher*.
>
> ---
>
> **Note:** New in version 2.8.
>
> ---

name
> The name of this partition.
>
> > **Type** *str*

scheduler
> The type of the backend scheduler of this partition.
>
> > **Returns** a subclass of *reframe.core.schedulers.Job*.
>
> ---
>
> **Note:** Changed in version 2.8.
>
> Prior versions returned a string representing the scheduler and job launcher combination.
>
> ---

class reframe.core.schedulers.**Job**(*name*, *command*, *launcher*, *environs=[]*, *workdir='.'*, *num_tasks=1*, *num_tasks_per_node=None*, *num_tasks_per_core=None*, *num_tasks_per_socket=None*, *num_cpus_per_task=None*, *use_smt=None*, *time_limit=(0, 10, 0)*, *script_filename=None*, *stdout=None*, *stderr=None*, *sched_account=None*, *sched_partition=None*, *sched_reservation=None*, *sched_nodelist=None*, *sched_exclude_nodelist=None*, *sched_exclusive_access=None*, *sched_options=[]*)

Bases: abc.ABC

A job descriptor.

Note: This is an abstract class. Users may not create jobs directly.

---

launcher
> The parallel program launcher that will be used to launch the parallel executable of this job.
>
> > **Type** *reframe.core.launchers.JobLauncher*

options
> Options to be passed to the backend job scheduler.
>
> > **Type** list of str
>
> > **Default** []

post_run
> List of shell commands to execute after launching this job.
>
> See *pre_run* for a more detailed description of the semantics.

> **Type** list of str
>
> **Default** []

pre_run
> List of shell commands to execute before launching this job.
>
> These commands do not execute in the context of ReFrame. Instead, they are emitted in the generated job script just before the actual job launch command.
>
> > **Type** list of str
> >
> > **Default** []

class reframe.core.launchers.**JobLauncher**(*options=[]*)
> Bases: abc.ABC
>
> A job launcher.
>
> A job launcher is the executable that actually launches a distributed program to multiple nodes, e.g., mpirun, srun etc.
>
> ---
>
> **Note:** This is an abstract class. Regression tests may not instantiate this class directly.
>
> ---
>
> ---
>
> **Note:** Changed in version 2.8: Job launchers do not get a reference to a job during their initialization.
>
> ---
>
> **options**
> > List of options to be passed to the job launcher invocation.
> >
> > > **Type** list of str
> > >
> > > **Default** []

class reframe.core.launchers.**LauncherWrapper**(*target_launcher*, *wrapper_command*, *wrapper_options=[]*)
> Bases: *reframe.core.launchers.JobLauncher*
>
> Wrap a launcher object so as to modify its invocation.
>
> This is useful for parallel debuggers. For example, to launch a regression test using the DDT debugger, you can do the following:

```
def setup(self, partition, environ, **job_opts):
    super().setup(partition, environ, **job_opts)
    self.job.launcher = LauncherWrapper(self.job.launcher, 'ddt',
                                        ['--offline'])
```

> If the current system partition uses native Slurm for job submission, this setup will generate the following command in the submission script:

```
ddt --offline srun <test_executable>
```

> If the current partition uses mpirun instead, it will generate

```
ddt --offline mpirun -np <num_tasks> ... <test_executable>
```

> **Parameters**
> > • **target_launcher** – The launcher to wrap.

- **wrapper_command** – The wrapper command.

- **wrapper_options** – List of options to pass to the wrapper command.

reframe.core.launchers.registry.**getlauncher**(*name*)
 Get launcher by its registered name.

 The available names are those specified in the *configuration file*.

 This method may become handy in very special situations, e.g., testing an application that needs to replace the system partition launcher or if a different launcher must be used for a different programming environment.

 For example, if you want to replace the current partition's launcher with the local one, here is how you can achieve it:

```
def setup(self, partition, environ, **job_opts):
    super().setup(partition, environ, **job_opts)
    self.job.launcher = getlauncher('local')()
```

 Note that this method returns a launcher class type and not an instance of that class. You have to instantiate it explicitly before assigning it to the launcher attribute of the job.

> **Parameters name** – The name of the launcher to retrieve.
>
> **Returns** The class of the launcher requested, which is a subclass of *reframe.core.launchers.JobLauncher*.
>
> **Raises reframe.core.exceptions.ConfigError** – if no launcher is registered with that name.

 Note: New in version 2.8.

reframe.core.launchers.registry.**register_launcher**(*name*, *local=False*)
 Class decorator for registering new job launchers.

> **Parameters**
>
> - **name** – The registration name of this launcher
>
> - **local** – True if launcher may only submit local jobs, False otherwise.
>
> **Raises ValueError** – if a job launcher is already registered with the same name.

 Note: New in version 2.8.

 This method is only relevant to developers of new job launchers.

3.11 Sanity Functions Reference

Sanity deferrable functions.

This module provides functions to be used with the *sanity_patterns* and :attr'perf_patterns <reframe.core.pipeline.RegressionTest.perf_patterns>'. The key characteristic of these functions is that they are not executed the time they are called. Instead they are evaluated at a later point by the framework (inside the *check_sanity* and *check_performance* methods). Any sanity function may be evaluated either explicitly or implicitly.

3.11.1 Explicit evaluation of sanity functions

Sanity functions may be evaluated at any time by calling the `evaluate` on their return value.

3.11.2 Implicit evaluation of sanity functions

Sanity functions may also be evaluated implicitly in the following situations:

- When you try to get their truthy value by either explicitly or implicitly calling `bool` on their return value. This implies that when you include the result of a sanity function in an `if` statement or when you apply the `and`, `or` or `not` operators, this will trigger their immediate evaluation.

- When you try to iterate over their result. This implies that including the result of a sanity function in a `for` statement will trigger its evaluation immediately.

- When you try to explicitly or implicitly get its string representation by calling `str` on its result. This implies that printing the return value of a sanity function will automatically trigger its evaluation.

This module provides three categories of sanity functions:

1. Deferrable replacements of certain Python built-in functions. These functions simply delegate their execution to the actual built-ins.

2. Assertion functions. These functions are used to assert certain conditions and they either return `True` or raise `reframe.core.exceptions.SanityError` with a message describing the error. Users may provide their own formatted messages through the `msg` argument. For example, in the following call to *assert_eq()* the `{0}` and `{1}` placeholders will obtain the actual arguments passed to the assertion function.

```
assert_eq(a, 1, msg="{0} is not equal to {1}")
```

If in the user provided message more placeholders are used than the arguments of the assert function (except the `msg` argument), no argument substitution will be performed in the user message.

3. Utility functions. The are functions that you will normally use when defining *sanity_patterns* and *perf_patterns*. They include, but are not limited to, functions to iterate over regex matches in a file, extracting and converting values from regex matches, computing statistical information on series of data etc.

`reframe.utility.sanity.`**`abs`**(*x*)
> Replacement for the built-in `abs()` function.

`reframe.utility.sanity.`**`all`**(*iterable*)
> Replacement for the built-in `all()` function.

`reframe.utility.sanity.`**`and_`**(*a*, *b*)
> Deferrable version of the `and` operator.

> > **Returns** `a and b`.

`reframe.utility.sanity.`**`any`**(*iterable*)
> Replacement for the built-in `any()` function.

`reframe.utility.sanity.`**`assert_bounded`**(*val*, *lower=None*, *upper=None*, *msg=None*)
> Assert that `lower <= val <= upper`.

> > **Parameters**

> > - **val** – The value to check.

> > - **lower** – The lower bound. If `None`, it defaults to `-inf`.

> > - **upper** – The upper bound. If `None`, it defaults to `inf`.

Returns True on success.

Raises **reframe.core.exceptions.SanityError** – if assertion fails.

reframe.utility.sanity.**assert_eq**(*a*, *b*, *msg=None*)
Assert that a == b.

Returns True on success.

Raises **reframe.core.exceptions.SanityError** – if assertion fails.

reframe.utility.sanity.**assert_false**(*x*, *msg=None*)
Assert that x is evaluated to False.

Returns True on success.

Raises **reframe.core.exceptions.SanityError** – if assertion fails.

reframe.utility.sanity.**assert_found**(*patt*, *filename*, *msg=None*, *encoding='utf-8'*)
Assert that regex pattern patt is found in the file filename.

Parameters

- **patt** – The regex pattern to search. Any standard Python regular expression is accepted.
- **filename** – The name of the file to examine. Any OSError raised while processing the file will be propagated as a reframe.core.exceptions.SanityError.
- **encoding** – The name of the encoding used to decode the file.

Returns True on success.

Raises **reframe.core.exceptions.SanityError** – if assertion fails.

reframe.utility.sanity.**assert_ge**(*a*, *b*, *msg=None*)
Assert that a >= b.

Returns True on success.

Raises **reframe.core.exceptions.SanityError** – if assertion fails.

reframe.utility.sanity.**assert_gt**(*a*, *b*, *msg=None*)
Assert that a > b.

Returns True on success.

Raises **reframe.core.exceptions.SanityError** – if assertion fails.

reframe.utility.sanity.**assert_in**(*item*, *container*, *msg=None*)
Assert that item is in container.

Returns True on success.

Raises **reframe.core.exceptions.SanityError** – if assertion fails.

reframe.utility.sanity.**assert_le**(*a*, *b*, *msg=None*)
Assert that a <= b.

Returns True on success.

Raises **reframe.core.exceptions.SanityError** – if assertion fails.

reframe.utility.sanity.**assert_lt**(*a*, *b*, *msg=None*)
Assert that a < b.

Returns True on success.

Raises **reframe.core.exceptions.SanityError** – if assertion fails.

`reframe.utility.sanity.`**`assert_ne`**(*a*, *b*, *msg=None*)
    Assert that `a != b`.

        Returns `True` on success.

        Raises `reframe.core.exceptions.SanityError` – if assertion fails.

`reframe.utility.sanity.`**`assert_not_found`**(*patt*, *filename*, *msg=None*, *encoding='utf-8'*)
    Assert that regex pattern `patt` is not found in the file `filename`.

    This is the inverse of *assert_found()*.

        Returns `True` on success.

        Raises `reframe.core.exceptions.SanityError` – if assertion fails.

`reframe.utility.sanity.`**`assert_not_in`**(*item*, *container*, *msg=None*)
    Assert that `item` is not in `container`.

        Returns `True` on success.

        Raises `reframe.core.exceptions.SanityError` – if assertion fails.

`reframe.utility.sanity.`**`assert_reference`**(*val*, *ref*, *lower_thres=None*, *upper_thres=None*, *msg=None*)
    Assert that value `val` respects the reference value `ref`.

        Parameters

            • **`val`** – The value to check.

            • **`ref`** – The reference value.

            • **`lower_thres`** – The lower threshold value expressed as a negative decimal fraction of the reference value. Must be in [-1, 0]. If `None`, no lower thresholds is applied.

            • **`upper_thres`** – The upper threshold value expressed as a decimal fraction of the reference value. Must be in [0, 1]. If `None`, no upper thresholds is applied.

        Returns `True` on success.

        Raises `reframe.core.exceptions.SanityError` – if assertion fails or if the lower and upper thresholds do not have appropriate values.

`reframe.utility.sanity.`**`assert_true`**(*x*, *msg=None*)
    Assert that `x` is evaluated to `True`.

        Returns `True` on success.

        Raises `reframe.core.exceptions.SanityError` – if assertion fails.

`reframe.utility.sanity.`**`avg`**(*iterable*)
    Return the average of all the elements of `iterable`.

`reframe.utility.sanity.`**`chain`**(**iterables*)
    Replacement for the `itertools.chain()` function.

`reframe.utility.sanity.`**`contains`**(*seq*, *key*)
    Deferrable version of the `in` operator.

        Returns `key in seq`.

`reframe.utility.sanity.`**`count`**(*iterable*)
    Return the element count of `iterable`.

    This is similar to the built-in `len()`, except that it can also handle any argument that supports iteration, including generators.

`reframe.utility.sanity.`**`enumerate`**(*iterable*, *start=0*)
    Replacement for the built-in `enumerate()` function.

`reframe.utility.sanity.`**`extractall`**(*patt*, *filename*, *tag=0*, *conv=None*, *encoding='utf-8'*)
    Extract all values from the capturing group `tag` of a matching regex `patt` in the file `filename`.

>   **Parameters**
>
>   - **`patt`** – The regex pattern to search. Any standard Python regular expression is accepted.
>
>   - **`filename`** – The name of the file to examine.
>
>   - **`encoding`** – The name of the encoding used to decode the file.
>
>   - **`tag`** – The regex capturing group to be extracted. Group `0` refers always to the whole match. Since the file is processed line by line, this means that group `0` returns the whole line that was matched.
>
>   - **`conv`** – A callable that takes a single argument and returns a new value. If provided, it will be used to convert the extracted values before returning them.
>
>   **Returns** A list of the extracted values from the matched regex.
>
>   **Raises** `reframe.core.exceptions.SanityError` – In case of errors.

`reframe.utility.sanity.`**`extractiter`**(*patt*, *filename*, *tag=0*, *conv=None*, *encoding='utf-8'*)
    Get an iterator over the values extracted from the capturing group `tag` of a matching regex `patt` in the file `filename`.

    This function is equivalent to `extractall()` except that it returns a generator object, instead of a list, which you can use to iterate over the extracted values.

`reframe.utility.sanity.`**`extractsingle`**(*patt*, *filename*, *tag=0*, *conv=None*, *item=0*, *encoding='utf-8'*)
    Extract a single value from the capturing group `tag` of a matching regex `patt` in the file `filename`.

    This function is equivalent to `extractall(patt, filename, tag, conv)[item]`, except that it raises a `SanityError` if `item` is out of bounds.

>   **Parameters**
>
>   - **`patt`** – as in `extractall()`.
>
>   - **`filename`** – as in `extractall()`.
>
>   - **`encoding`** – as in `extractall()`.
>
>   - **`tag`** – as in `extractall()`.
>
>   - **`conv`** – as in `extractall()`.
>
>   - **`item`** – the specific element to extract.
>
>   **Returns** The extracted value.
>
>   **Raises** `reframe.core.exceptions.SanityError` – In case of errors.

`reframe.utility.sanity.`**`filter`**(*function*, *iterable*)
    Replacement for the built-in `filter()` function.

`reframe.utility.sanity.`**`findall`**(*patt*, *filename*, *encoding='utf-8'*)
    Get all matches of regex `patt` in `filename`.

>   **Parameters**
>
>   - **`patt`** – The regex pattern to search. Any standard Python regular expression is accepted.
>
>   - **`filename`** – The name of the file to examine.

- **encoding** – The name of the encoding used to decode the file.

> **Returns** A list of raw regex match objects.
>
> **Raises** **reframe.core.exceptions.SanityError** – In case an OSError is raised while processing filename.

reframe.utility.sanity.**finditer**(*patt*, *filename*, *encoding='utf-8'*)

> Get an iterator over the matches of the regex patt in filename.
>
> This function is equivalent to *findall()* except that it returns a generator object instead of a list, which you can use to iterate over the raw matches.

reframe.utility.sanity.**getattr**(*obj*, *attr*, **args*)

> Replacement for the built-in getattr() function.

reframe.utility.sanity.**getitem**(*container*, *item*)

> Get item from container.
>
> container may refer to any container that can be indexed.
>
> **Raises** **reframe.core.exceptions.SanityError** – In case item cannot be retrieved from container.

reframe.utility.sanity.**glob**(*pathname*, ***, *recursive=False*)

> Replacement for the glob.glob() function.

reframe.utility.sanity.**hasattr**(*obj*, *name*)

> Replacement for the built-in hasattr() function.

reframe.utility.sanity.**iglob**(*pathname*, *recursive=False*)

> Replacement for the glob.iglob() function.

reframe.utility.sanity.**len**(*s*)

> Replacement for the built-in len() function.

reframe.utility.sanity.**map**(*function*, **iterables*)

> Replacement for the built-in map() function.

reframe.utility.sanity.**max**(**args*)

> Replacement for the built-in max() function.

reframe.utility.sanity.**min**(**args*)

> Replacement for the built-in min() function.

reframe.utility.sanity.**not_**(*a*)

> Deferrable version of the not operator.
>
> **Returns** not a.

reframe.utility.sanity.**or_**(*a*, *b*)

> Deferrable version of the or operator.
>
> **Returns** a or b.

reframe.utility.sanity.**reversed**(*seq*)

> Replacement for the built-in reversed() function.

reframe.utility.sanity.**round**(*number*, **args*)

> Replacement for the built-in round() function.

reframe.utility.sanity.**sanity_function**(*func*)

> **Decorator** Sanity function decorator.
>
> Decorate any function to be used in sanity and/or performance patterns with this decorator:

```
@sanity_function
def myfunc(*args):
    do_sth()
```

This decorator is an alias to the `reframe.core.deferrable.deferrable()` decorator. The following function definition is equivalent to the above:

```
@deferrable
def myfunc(*args):
    do_sth()
```

`reframe.utility.sanity.`**`setattr`**(*obj*, *name*, *value*)
    Replacement for the built-in `setattr()` function.

`reframe.utility.sanity.`**`sorted`**(*iterable*, **args*)
    Replacement for the built-in `sorted()` function.

`reframe.utility.sanity.`**`sum`**(*iterable*, **args*)
    Replacement for the built-in `sum()` function.

`reframe.utility.sanity.`**`zip`**(**iterables*)
    Replacement for the built-in `zip()` function.

Python Module Index

r

Index