Cosmic Ray Documentation Documentation

Austin Bingham

Mar 14, 2023

Contents

> "Four human beings – changed by space-born cosmic rays into something more than merely human."
>
> —The Fantastic Four

Cosmic Ray is a mutation testing tool for Python 3. It makes small changes to your production source code, running your test suite for each change. If a test suite passes on mutated code, then you have a mismatch between your tests and your functionality.

Like coverage analysis, mutation testing helps ensure that you're testing all of your code. But while coverage only tells you if a line of code is executed, mutation testing will determine if your tests actually check the behavior of your code. This adds tremendous value to your test suite by helping it fulfill its primary role: making sure your code does what you expect it to do!

Cosmic Ray has been successfully used on a wide variety of projects ranging from assemblers to oil exploration software.

CHAPTER 1

Contents

1.1 Theory

Mutation testing is conceptually simple and elegant. You make certain kinds of controlled changes (mutations) to your *code under test*[1], and then you run your test suite over this mutated code. If your test suite fails, then we say that your tests "killed" (i.e. detected) the mutant. If the changes cause your code to simply crash, then we say the mutant is "incompetent". If your test suite passes, however, we say that the mutant has "survived".

Needless to say, we want to kill all of the mutants.

The goal of mutation testing is to verify that your test suite is actually testing all of the parts of your code that it needs to, and that it is doing so in a meaningful way. If a mutant survives your test suite, this is an indication that your test suite is not adequately checking the code that was changed. This means that either a) you need more or better tests or b) you've got code which you don't need.

You can read more about mutation testing at the repository of all human knowledge. Lionel Brian has a nice set of slides introducing mutation testing as well.

1.2 Tutorial: The basics

This tutorial will walk you through the steps needed to install, configure, and run Cosmic Ray.

1.2.1 Installation

First you'll need to install Cosmic Ray. The simplest (and generally best) way to do this is with `pip`:

```
pip install cosmic-ray
```

You'll generally want to do this in a virtual environment, but it's not required.

[1] By "code under test", we mean the code that your test suite is testing. Mutation testing is trying to ensure that your unaltered test suite can detect explicitly incorrect behavior in your code.

Installation from source

If you need to install Cosmic Ray from source, first change to the directorying containing `setup.py`. Then run:

```
pip install .
```

Or, if you want to install from source in "editable" mode, you can use the "-e" flag:

```
pip install -e .
```

1.2.2 Source module and tests

Mutation testing works by making small mutations to the *code under test* (CUT) and then running a test suite over the mutated code. For this tutorial, then, we'll need to create our CUT and a test suite for it.

You should create a new directory which will contain the CUT, the tests, and eventually the Cosmic Ray data. For the rest of this tutorial we'll refer to this new directory as `ROOT` (or `$ROOT` if we're showing shell code).

Now create the file `ROOT/mod.py` with these contents:

```
def func():
    return 1234
```

This file contains your code under test, i.e. the code that Cosmic Ray will mutate. It's clearly very simple, and it has very few opportunities for mutation, but it's sufficient for this tutorial. In fact, having simple code like this will make it easier to see what Cosmic Ray is doing without getting bogged down by scale.

Next create the file `ROOT/test_mod.py` with these contents:

```
import unittest
import mod


class Tests(unittest.TestCase):
    def test_func(self):
        self.assertEqual(mod.func(), 1234)
```

This contains the test suite for `mod.py`. Cosmic Ray will not mutate this code. Rather, it will run this test suite for every mutation that it creates.

Before moving on, let's make sure that the test suite works correctly:

```
python -m unittest test_mod.py
```

This should show that all tests pass:

```
.
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

If you see one test passing like this, then you're ready to continue!

1.2.3 Creating a configuration

Before you do run any mutation tests, you need to create a *configuration*. A configuration is a TOML file that specifies the modules you want to mutate, the test scripts to use, and so forth. A configuration is used to create a *session*, something we'll look at in the next section.

The `new-config` command

You can create a configuration by hand if you want. In fact, you'll generally need to edit them by hand to get the exact configuration you need. But you can create an initial configuration using the `new-config` command. This will ask you a series of questions and construct a new configuration based on your answers.

To create your config for this tutorial, do this:

```
cd $ROOT
cosmic-ray new-config tutorial.toml
```

This will ask you a series of questions. Answer them like this:

```
[?] Top-level module path: mod.py
[?] Test execution timeout (seconds): 10
[?] Test command: python -m unittest test_mod.py
-- MENU: Distributor --
  (0) http
  (1) local
[?] Enter menu selection: 1
```

This will create the file `tutorial.toml` with these contents:

```
1 [cosmic-ray]
2 module-path = "mod.py"
3 timeout = 10.0
4 excluded-modules = []
5 test-command = "python -m unittest test_mod.py"
6
7 [cosmic-ray.distributor]
8 name = "local"
```

Configuration contents

Let's examine the contents of this file before moving on. On line 1 we define the 'cosmic-ray' key in the TOML structure; this key will contain all Cosmic Ray configuration information.

On line 2 we set the 'module-path' key to the string "mod.py":

```
module-path = "mod.py"
```

This tells Cosmic Ray that we're going to be mutating the module in the file `mod.py`. Every Cosmic Ray configuration refers to a single top-level module that will be mutated, and in this case we're telling Cosmic Ray to mutate the `mod` module, contained in the file `mod.py`.

Note: The 'module-path' is a *path* to a file or directory, not the name of the module of package. If it's a file then Cosmic Ray will treat it as a single module, but if it's a directory then Cosmic Ray will treat it as a package.

When working on a package, Cosmic Ray will apply mutations to all submodules in the package.

Additionally, the 'module-path' can be a list of directories or files: *module-path = ["file1.py", "some_directory"]*

Line 3 tells Cosmic Ray the maximium amount of time to let a test run before it's considered a failure:

```
timeout = 10.0
```

In this case, we're telling Cosmic Ray to kill a test if it runs longer than 10 seconds. This timeout is important because some mutations can cause the tests to go into an infinite loop. Without timeout we'd never exit the test! It's important to set this timeout such that it's long enough for all legitimate tests.

Next, line 4 tells Cosmic Ray which modules to exclude from mutation:

```
excluded-modules = []
```

In this case we're not excluding any, but there may be times when you need to skip certain modules, e.g. because you know that you don't have sufficient tests for them at the moment. This parameter expects glob-patterns, so to exclude files that end with `_test.py` recursively for example, you would add `"**/*_test.py"`.

Line 5 is one of the most critical lines in the configuration. This tells Cosmic Ray how to run your test suite:

```
test-command = "python -m unittest test_mod.py"
```

In this case, our test suite uses the standard unittest testing framework, and the tests are in the file `test_mod.py`.

The last two lines tell Cosmic Ray which "distributor" to use:

```
[cosmic-ray.distributor]
name = "local"
```

A distributor controls how mutation jobs are assigned to one or more workers so that they can (potentially) run in parallel. In this case we're using the default 'local' distributor which only runs one mutation at a time. There are other, more sophisticated distributors which we discuss elsewhere.

1.2.4 Create a session and baseline

Cosmic Ray uses a notion of *sessions* to encompass a full mutation testing suite. Since mutation testing runs can take a long time, and since you might need to stop and start them, sessions store data about the progress of a run.

Note: Most Cosmic Ray commands allow you to increase their "verbosity" via the command line. This will make them print out more information about what they're doing.

Try adding "–verbosity INFO" to the command you run if you more details about what's going on!

Initializing a session

The first step in a full testing run, then, is to initialize a session:

```
cosmic-ray init tutorial.toml tutorial.sqlite
```

Note: This command prepares all the mutations that will later be applied to code. As such, its execution time is proportional to the amount of code and the code complexitly. You can expect about 15-30s per 1kloc.

This will create a database file called `tutorial.sqlite`. There is a record in the database for each mutation that Cosmic Ray will perform, and Cosmic Ray will associate testing results with these records as it executes.

When does *init* need to be run?

init completely rewrites the session file you tell it to use, so you should not re-run *init* on a session that contains any results that you want to keep. At the same time, if you change your configuration in a way that alters which tests are run and which mutations are made, then you should re-initialize your session.

Generally speaking, if you change the 'module-path', 'timeout', 'excluded-modules', or 'test-command' parts of your configuration, or if you change any of the filters you use, then you need to re-initialize your session and start over. Any of these changes can affect the operations that the subsequent *exec* command will run.

Similarly, you need to create a new session with *init* whenever your code-under-test or your tests themselves change. This is necessary because changes to the CUT will affect which mutations are made and changes to the tests affect which tests are run.

Baselining

Before running the full mutation suite, it's important to make sure that the test suite passes in the absence of any mutations. If the test suite does *not* pass in the absence of mutations, then the results of the mutation testing are essentially useless.

You can use the `baseline` command to check that the test suite passes on unmutated code:

```
cosmic-ray --verbosity=INFO baseline tutorial.toml
```

This should report that the tests pass, something like this:

```
[07/23/21 10:00:20] INFO     INFO:root:Reading config from 'tutorial.toml'
↪                                                                        ␣
↪                                                  config.py:103         ␣
                    INFO     INFO:cosmic_ray.commands.execute:Beginning execution
↪                                                                        ␣
↪                                                  execute.py:45         ␣
                    INFO     INFO:cosmic_ray.testing:Running test (timeout=10.0):␣
↪python -m unittest test_mod.py                                          ␣
↪                                                      testing.py:36     ␣
                    INFO     INFO:cosmic_ray.commands.execute:Job baseline complete
↪                                                                        ␣
↪                                                  execute.py:43         ␣
                    INFO     INFO:cosmic_ray.commands.execute:Execution finished
↪                                                                        ␣
↪                                                  execute.py:53         ␣
                    INFO     INFO:root:Baseline passed. Execution with no mutation␣
↪works fine.
```

If this command succeeds, then you're ready to start mutating code and testing it!

1.2.5 Examining the session with cr-report

Our session file, `tutorial.sqlite`, is essentially a list of mutations that Cosmic Ray will perform on the code under test. We haven't actually tested any mutants, so none of our mutations have testing results yet. With that in mind, let's examine the contents of our session with the `cr-report` program:

```
cr-report tutorial.sqlite --show-pending
```

This will produce output like this (though note that the test IDs will be different):

```
[job-id] f168ef23dff24b75846a730858fe0111
mod.py core/NumberReplacer 0
[job-id] 929a563b613242b48dae0f2de74ad2af
mod.py core/NumberReplacer 1
total jobs: 2
no jobs completed
```

This is telling us that Cosmic Ray detected two mutations that it can make to our code, both using the mutation operator "core/NumberReplacer". Without going into details, this means that Cosmic Ray has found one or more numeric literals in our code, and it plans to make two mutations to those numbers. We can see in our code that there is only one numeric literal, the value returned from `func()` on line 2:

```
1 def func():
2     return 1234
```

So Cosmic Ray is going to mutate that number in two ways, running the test suite each time.

The `cr-report` tool is useful for examining sessions, and it's main purpose is to give you summary reports after an entire session has been executed, which we'll do in the next step.

1.2.6 Execution

Now that you've initialized and baselined your session, it's time to start making mutants and testing them. We do this with the `exec` command. `exec` looks in your session file, `tutorial.sqlite`, for any mutations which were detected in the `init` phase that don't yet have results. For each of these, it performs the specified mutation and runs the test suite.

As we saw, we only have two mutations to make, and our test suite is very small. As a result the `exec` command will run quite quickly:

```
cosmic-ray exec tutorial.toml tutorial.sqlite
```

This should produce no output.

Note: The module and test suite for this tutorial are "toys" by design. As such, they run very quickly. Most real-world modules and test suites are much more substantial and require much longer to run. For example, if a test suite takes 10 seconds to run and Cosmic Ray finds 1000 mutations, a full `exec` will take 10 x 1000 = 10,000 seconds, or about 2.7 hours.

Committing before *exec*

If you're using revision control with your code (you are, right?!), you should consider committing your changes before running *exec*. While it's not strictly necessary to do this in simple cases, it's often important to commit if you're using tools like *cr-http-workers* that rely on fetching code from a repository.

Also, while Cosmic Ray is designed to be robust in the face of exceptions and crashes, there is always the possibility that Cosmic Ray won't correctly undo a mutation. Remember, it makes mutations directly on disk, so if a mutation is not correctly undone, and if you haven't committed your changes prior to testing, you run the risk of introducing a mutation into you code accidentally.

1.2.7 Reporting the results

Assuming it ran correctly, we can now use `cr-report` to see the updated state of our session:

```
cr-report tutorial.sqlite --show-pending
```

This time we see that both mutations were made, tests were run for each, and both were "killed":

```
[job-id] f168ef23dff24b75846a730858fe0111
mod.py core/NumberReplacer 0
worker outcome: normal, test outcome: killed
[job-id] 929a563b613242b48dae0f2de74ad2af
mod.py core/NumberReplacer 1
worker outcome: normal, test outcome: killed
total jobs: 2
complete: 2 (100.00%)
surviving mutants: 0 (0.00%)
```

Tip: You don't have to wait for `exec` to complete to generate a report. If you have a long-running session and want to see your progress, you can execute `cr-report` while `cosmic-ray exec` is running to view the progress the latter is making.

HTML reports

You can also generate a handy HTML report with *cr-html*:

```
cr-html tutorial.sqlite > report.html
```

You can then open `report.html` in your browser to see the details. One nice feature of these HTML reports is that they show the actual mutation that was used.

1.3 Tutorial: Distributed, concurrent mutation testing

One of the main practical challenges to mutation testing is that it can take a long time. Even on moderately sized projects, you might need millions of individual mutations and test runs. This can be prohibitive to run on a single system.

One way to cope with these long runtimes is to parallelize the mutation and testing procedures. Fortunately, mutation testing is embarassingly parallel in nature, so we can apply some relatively simple techniques to get really nice scaling up of the work. To support parallel execution of mutation testing runs, Cosmic Ray has the notion of *distributors* which can control where and how tests are run. Different distributors can run tests in different contexts: in parallel on a single machine, by distributing them across a message bus, or perhaps by spawning test runs on cloud systems.

1.3.1 The HTTP distributor

Cosmic Ray includes `cosmic_ray.distributors.http.HttpDistributor`, a distributor which allows you to send mutation-and-test requests to workers running locally or remotely. You can run as many of these workers as you want, thereby making it possible to run as many mutations in parallel as you want.

Each worker is a small HTTP server, listening for requests from the `exec` command to perform a mutation and test. Each worker handlers only one mutation request at a time. Critically, each worker has its own copy of the code under test, meaning that it can make mutations to that copy of the code without interfering with other workers.

You need to make sure that workers are running prior to running the `exec` command. `exec` doesn't have any support for starting workers. The major configuration involved with the HTTP distributor is telling `exec` where there workers are listening.

A sample project

To demonstrate `HttpDistributor` we'll need a sample module and test suite. We'll use a very simple set of code, as we did in *the basic tutorial*.

Create a new directory to hold this code. We'll refer to this directory as `ROOT`.

Create the file `ROOT/mod.py` with these contents:

```
def func():
    return 1234
```

Then create `ROOT/test_mod.py` with these contents:

```
import unittest
import mod


class Tests(unittest.TestCase):
    def test_func(self):
        self.assertEqual(mod.func(), 1234)
```

Finally, we'll create a configuration, `ROOT/config.toml`:

```
1  [cosmic-ray]
2  module-path = "mod.py"
3  timeout = 10.0
4  excluded-modules = []
5  test-command = "python -m unittest test_mod.py"
6
7  [cosmic-ray.distributor]
8  name = "http"
9
10 [cosmic-ray.distributor.http]
11 worker-urls = ["http://localhost:9876"]
```

This config is similar to others that we've looked at, with the major difference that it specifies the use of the 'http' distributor rather than 'local'. On line 8 we set "cosmic-ray.distributor.name" to "http".

Then on line 11 we set the "cosmic-ray.distributor.http.worker-urls" setting to a list containing a URL. This is the address at which a *worker* will be listening for mutation requests. This configuration only specifies a single worker, but we can put as many workers here as we want.

Starting a worker

Before Cosmic Ray can send requests to a worker, we need to start it. From the `ROOT` directory, start a worker using the `http-worker` command:

```
cd $ROOT
cosmic-ray --verbosity INFO http-worker --port 9876
```

The `--verbosity INFO` argument configures the worker's logging to show more messages than normal. The `--port 9876` argument instructs it to listen for requests on port 9876, the same port we specified in the 'worker-urls' list in our configuration. The worker will tell you that it's waiting to process requests on port 9876:

```
======== Running on http://0.0.0.0:9876 ========
(Press CTRL+C to quit)
```

Note that your worker must be running in the same directory as you would normally run the tests from. In this case, we're expecting the tests to be run in `$ROOT`, so make sure your worker is running in that directory. Generally speaking, the worker doesn't do much more than mutate the code on disk and run the test command you've specified in your config.

Running the tests

We need to leave the worker running in its own terminal, so for these next steps you'll need to start a new terminal.

First we need to initialize a new Cosmic Ray session:

```
cd $ROOT
cosmic-ray init config.toml session.sqlite
```

Once the session is created, we can execute the tests:

```
cosmic-ray exec config.toml session.sqlite
```

This should execute very quickly. The most important thing to note is that our worker process is where the mutation and testing actually occurred. If you switch back to the terminal hosting your worker, you should see that it produced output something like this:

```
[05/16/21 11:31:10] INFO     INFO:cosmic_ray.mutating:Applying mutation: path=mod.py,␣
↪                            mutating.py:111
                             op=<cosmic_ray.operators.number_replacer.NumberReplacer␣
↪object at 0x10d2b9550>,
                             occurrence=1
                    INFO     INFO:cosmic_ray.testing:Running test (timeout=10.0):␣
↪python -m unittest test_mod.py       testing.py:36
                    INFO     INFO:aiohttp.access:::1 [16/May/2021:09:31:10 +0000]
↪"POST / HTTP/1.1" 200 899 "-"       web_log.py:206
                             "Python/3.7 aiohttp/3.7.4.post0"
                    INFO     INFO:cosmic_ray.mutating:Applying mutation: path=mod.py,␣
↪                            mutating.py:111
                             op=<cosmic_ray.operators.number_replacer.NumberReplacer␣
↪object at 0x10d4cdf60>,
                             occurrence=0
                    INFO     INFO:cosmic_ray.testing:Running test (timeout=10.0):␣
↪python -m unittest test_mod.py       testing.py:36
[05/16/21 11:31:11] INFO     INFO:aiohttp.access:::1 [16/May/2021:09:31:10 +0000]
↪"POST / HTTP/1.1" 200 899 "-"       web_log.py:206
                             "Python/3.7 aiohttp/3.7.4.post0"
```

Congratulations! You've just performed your first distributed mutation testing with Cosmic Ray. There are other details you need to consider when scaling beyond a single worker, but this small example covers the most important elements: setting up the configuration and starting a worker.

At this point you can kill the worker you started earlier.

1.3.2 Concurrent execution with multiple workers

In the previous example we only ran a single worker process, so from a concurrency point of view this was no different from using the 'local' distributor. Before we can run multiple workers, though, we need to consider what resources each worker requires. Ultimately, each worker needs two things:

- An HTTP endpoint
- A copy of the code under test that it can modify

In this example we'll create the unique endpoints by giving each worker its own port. In principle, though, workers may be running on entirely different machines on a network.

Distinct copies of the code

As mentioned earlier, Cosmic Ray mutation works by actually modifying the code on disk. As such, multiple workers can't share a single copy of the code; their mutations would interfere with one another. So we need to make sure each worker has a copy of the code under test.

For this example, we'll manually copy the files around:

```
cd $ROOT
mkdir worker1
cp mod.py worker1
cp test_mod.py worker1
mkdir worker2
cp mod.py worker2
cp test_mod.py worker2
```

Now the directories `worker1` and `worker2` contain separate copies of the code under test.

Starting the workers

Now we can start the workers. Remember that each will run in its own terminal. In one terminal, start the first worker:

```
cd $ROOT/worker1
cosmic-ray --verbosity INFO http-worker --port 9876
```

Then in another terminal start a second worker:

```
cd $ROOT/worker2
cosmic-ray --verbosity INFO http-worker --port 9877
```

Note that the workers are using different ports.

Update the configuration

To tell Cosmic Ray to use both of these workers, we need to update our configuration. Edit `config.toml` to specify both workers URLs:

```
1  [cosmic-ray]
2  module-path = "mod.py"
3  timeout = 10.0
4  excluded-modules = []
5  test-command = "python -m unittest test_mod.py"
6
7  [cosmic-ray.distributor]
8  name = "http"
9
10 [cosmic-ray.distributor.http]
11 worker-urls = ["http://localhost:9876", "http://localhost:9877"]
```

On line 11 we now list the endpoints for both workers.

Running the tests

We're now ready to run the tests. Go back to `ROOT` and re-initialize your session:

```
cd $ROOT
cosmic-ray init config.toml session.sqlite
```

Finally, we can execute the tests:

```
cosmic-ray exec config.toml session.sqlite
```

If you run `cr-report` you should see that two tests were run and that there were no survivors:

```
$ cr-report session.sqlite
e4e56a71a059466f861d62c987988efe mod.py core/NumberReplacer 0
worker outcome: normal, test outcome: killed
7820da3f68cd40a7b60d69506e87c4aa mod.py core/NumberReplacer 1
worker outcome: normal, test outcome: killed
total jobs: 2
complete: 2 (100.00%)
surviving mutants: 0 (0.00%)
```

Likewise, if you look at the terminals for your two workers, you should see that they each received a request to perform a mutation test.

That's really all there is to distributed mutation testing with `HttpDistributor`. You simply start as many workers as you need, specifying their endpoints in your configuration.

Important: At this point you should kill the workers you started.

1.3.3 cr-http-workers: A tool for starting workers

It's extremely common for the code under test (and the tests themselves) to be in a git repository. As such, a simple way to create the isolated copies of the code that each worker requires is to clone this git repository. Once the mutation testing is done, these clones can be deleted.

To simplify this process Cosmic Ray provides `cr-http-workers`. This program reads your configuration to determine how many workers to start, and you provide it with a git repository to clone. For each 'worker-url' in your configuration it will clone the git repository and start a worker in that clone. You can then run `exec` to distribute work

to those workers. Once the testing is over, you can kill `cr-http-workers` and it will clean up the workers and their clones.

Preparing the git repository

To use `cr-http-workers` we first need a git repository, so we'll create one from our existing code.

Note: You should first delete the `worker1` and `worker2` directories if they still exist. This isn't critical, but it might be confusing to leave them around.

Here's how to initialize the git repository:

```
cd $ROOT
git init
git add mod.py
git add test_mod.py
git commit -a -m "initialized repo"
```

Running the workers

Once the git repo is initialized, we can start the workers:

```
cr-http-workers config.toml .
```

This tell `cr-http-workers` to read `config.toml` to determine the worker endpoints. The second argument, ".", tells it to clone the git repository in the current directory. In practice this repo URL will often be hosted elsewhere (e.g. github), but for our purposes we'll just work with the local repo.

This will start both workers processes, and the output from those workers will be shown in the output from `cr-http-workers`.

Running the tests

Once the workers are running, running the tests just involves the standard `init` and `exec` commands:

```
cd $ROOT
cosmic-ray init config.toml session.sqlite
cosmic-ray exec config.toml session.sqlite
```

Remember that you'll need to run this in another terminal.

Once the tests complete you can kill the `cr-http-workers` process. There's not much more to it than that!

Limitations

The main limitation of `cr-http-workers` is that it can only start workers on your local machine. If you want to run workers on other machines, you'll need to use some other mechanism. But very often, being able to run multiple workers on a single machine is a huge gain for mutation testing. Mutation testing time will scale down linearly with the number of workers you run, so running 4 workers on your system will - within certain limits - let you run your mutation testing 4 times faster.

1.3.4 Alternatives to HttpDistributor

If `HttpDistributor` doesn't meet your needs, Cosmic Ray allows you to write your own distributor and use it as a plugin. You might want to write a distributor plugin using Celery, for example, to take advantage of its sophisticated message bus.

1.4 Concepts

Cosmic Ray comprises a number of important and potentially confusing concepts. In this section we'll look at each of these concepts, explaining their role in Cosmic Ray and how they relate to other concepts. We'll also use this section to establish the terminology that we'll use throughout the rest of the documentation.

1.4.1 Operators

An *operator* in Cosmic Ray is a class that represents a specific type of mutation. The first role of an operator is to identify points in the code where a specific mutation can be applied. The second role of an operator is to actually perform the mutation when requested.

An example of an operator is `cosmic_ray.operators.break_continue`. As its name implies, this operator mutates code by replacing `break` with `continue`. During the initialization of a session, this operator identifies all of the locations in the code where this mutation can be applied. Then, during execution of a session, it actually mutates the code by replacing `break` nodes with `continue` nodes.

Operators are exposed to Cosmic Ray via plugins, and users can choose to extend the available operator set by providing their own operators. Operators are implemented as subclasses of *`cosmic_ray.operators.operator.Operator`*.

1.4.2 Distributors

Distributors determine the context in which tests are executed. The primary examples of distributors are `cosmic_ray.distribution.local.LocalDistributor` and `cosmic_ray.distribution.http.HttpDistributor`. The local distributor tests on the local machine, modifying an existing copy of the code in-place, running each test serially with no concurrency.

The http distributor distributes tests to remote workers via HTTP. There can be any number of workers, and they can run the tests in parallel. Because of this concurrency, each HTTP worker will generally have its own copy of the code under test.

Distributors have broad control over how they execute tests. During the execution phase they are given a sequence of pending mutations to execute, and it's their job to execute the tests in the appropriate context and return a result. Cosmic Ray doesn't impose any real constraints on how distributors accomplish this.

Distributors can require arbitrarily complex infrastructure and configuration. For example, the HTTP distributor requires you to start the workers prior to starting execution, and it requires that you provide each worker with its own copy of the code under test.

Distributors are implemented as plugins to Cosmic Ray. They are dynamically discovered, and users can create their own distributors. Cosmic Ray includes two execution engines plugins, *local* and *http*.

1.4.3 Configurations

A *configuration* is a TOML file that describes the work that Cosmic Ray will do. For example, it tells Cosmic Ray which modules to mutate, how to run tests, which tests to run, and so forth. You need to create a config before doing

any real work with Cosmic Ray.

You can create a skeleton config by running `cosmic-ray new-config <config file>`. This will ask you a series of questions and create a config from the answers. Note that this config will generally be incomplete and require you to edit it for completeness.

In many Cosmic Ray examples we'll use the name "config.toml" for configurations. You are not required to use this name, however. You can use any file name you want for your configurations.

Important: The full set of configuration options are not currently well documented. Each plugin can, in principle and often in practice, use their own specialized configuration options. We need to work on making the documentation of these options automatic and part of the plugin API. For detail on configuration options, the best place to check is currently in the `tests/example_project` directory.

1.4.4 Sessions

Cosmic Ray has a notion of *sessions* which encompass an entire mutation testing run. Essentially, a session is a database which records the work that needs to be done for a run. Then as results are available from workers that do the actual testing, the database is updated with results. By having a database like this, Cosmic Ray can safely stop in the middle of a (potentially very long) session and be restarted. Since the session knows which work is already completed, it can continue where it left off.

Sessions also allow for arbitrary post-facto analysis and report generation.

Initializing sessions

Before you can do mutation testing with Cosmic Ray, you need to first initialize a session. You can do this using the `init` command. With this command you tell Cosmic Ray a) the name of the session, b) which module(s) you wish to mutate and c) the location of the test suite. For example, to mutate the package `allele`, using the `unittest` to run the tests in `allele_tests`, and using the `local` execution engine, you could first need to create a configuration like this:

```
[cosmic-ray]
module-path = "allele"
timeout = 10
excluded-modules = []
test-command = python -m unittest allele_tests
distributor.name = "local"
```

You would run `cosmic-ray init` like this:

```
cosmic-ray init config.toml session.sqlite
```

You'll notice that this creates a new file called `allele_session.sqlite`. This is the database for your session.

1.4.5 Test suite

To be able to kill the mutants Cosmic Ray uses your test cases. But the mutants are not considered "more dead" when more test cases fail. Given that a single failing test case is sufficient to kill a mutant, it's a good idea to configure the test runner to exit as soon as a failing test case is found.

For `pytest` and `nose` that can be achieved with the `-x` option.

An important note on separating tests and production code

Cosmic Ray has a relatively simple view of how to mutate modules. Fundamentally, it will attempt to mutate any and all code in a module. This means that if you have test code in the same module as your code under test, Cosmic Ray will happily mutate the test code along with the production code. This is probably not what you want.

The best way to avoid this problem is to keep your test code in separate modules from your production code. This way you can tell Cosmic Ray precisely what to mutate.

Ideally, your test code will be in a different package from your production code. This way you can tell Cosmic Ray to mutate an entire package without needing to filter anything out. However, if your test code is in the same package as your production code (a common configuration), you can use the `excluded-modules` setting in your configuration to prevent mutation of your tests.

Given the choice, though, we recommend keeping your tests outside of the package for your code under test.

Executing tests

Once a session has been initialized, you can start executing tests by using the `exec` command. This command needs the config and the session you provided to `init`:

```
cosmic-ray exec config.toml session.sqlite
```

Normally this won't produce any output unless there are errors.

Viewing the results

Once your tests have completed, you can view the results using the `cr-report` command:

```
cr-report test_session.sqlite
```

This will give you detailed information about what work was done, followed by a summary of the entire session.

1.4.6 Test commands

The `test-command` field of a configuration tells Cosmic Ray how to run tests. Cosmic Ray runs this command from whatever directory you run the `exec` command (or, in the case of remote execution, in whatever directory the remote command handler is running).

1.4.7 Timeouts

One difficulty mutation testing tools have to face is how to deal with mutations that result in infinite loops (or other pathological runtime effects). Cosmic Ray takes the simple approach of using a *timeout* to determine when to kill a test and consider it *incompetent*. That is, if a test of a mutant takes longer than the timeout, the test is killed, and the mutant is marked incompetent.

You specify a test time through the `timeout` configuration key. This key specifies an absolute number of seconds that a test will be allowed to run. After the timeout is up, the test is killed. For example, to specify that tests should timeout after 10 seconds, use:

```
# config.toml
[cosmic-ray]
timeout = 10
```

1.5 How-tos

1.5.1 Filters

The `cosmic-ray init` commands scans a module for all possible mutations, but we don't always want to execute all of these. For example, we may know that some of these mutations will result in *equivalent mutants*, so we need a way to prevent these mutations from actually being run.

To account for this, Cosmic Ray includes a number of *filters*. Filters are nothing more than programs - generally small ones - that modify a session in some way, often by marking certains mutations as "skipped", thereby preventing them from running. The name "filter" is actually a bit misleading since these programs could modify a session in ways other than simply skipping some mutations. In practice, though, the need to skip certain tests is by far the most common use of these programs.

Using filters

Generally speaking, filters will be run immediately after running `cosmic-ray init`. It's up to you to decide which to run, and often they will be run along with `init` in a batch script or CI configuration.

For example, if you wanted to apply the `cr-filter-pragma` filter to your session, you could do something like this:

```
cosmic-ray init cr.conf session.sqlite
cr-filter-pragma session.sqlite
```

The `init` would first create a session where *all* mutation would be run, and then the `cr-filter-pragma` call would mark as skipped all mutations which are on a line with the pragma comment.

Filters included with Cosmic Ray

Cosmic Ray comes with a number of filters. Remember, though, that they are nothing more than simple programs that modify a session in some way; it should be straightforward to write your own filters should the need arise.

cr-filter-operators

`cr-filter-operators` allows you to filter out operators according to their names. You provide the filter with a set of regular expressions, and any Cosmic Ray operator who's name matches a one of these expressions will be skipped entirely.

The configuration is provided through a TOML file such as a standard Cosmic Ray configuration. The expressions must be in a list at the key "cosmic-ray.filters.operators-filter.exclude-operators". Here's an example:

```
[cosmic-ray.filters.operators-filter]
exclude-operators = [
  "core/ReplaceComparisonOperator_Is(Not)?_(Not)?(Eq|[LG]tE?)",
  "core/ReplaceComparisonOperator_(Not)?(Eq|[LG]tE?)_Is(Not)?",
```

(continues on next page)

(continued from previous page)

```
  "core/ReplaceComparisonOperator_LtE_Eq",
  "core/ReplaceComparisonOperator_Lt_NotEq",
]
```

The first regular expression here is skipping the following operators:

- core/ReplaceComparisonOperator_Is_Eq
- core/ReplaceComparisonOperator_Is_Lt
- core/ReplaceComparisonOperator_Is_LtE
- core/ReplaceComparisonOperator_Is_Gt
- core/ReplaceComparisonOperator_Is_GtE
- core/ReplaceComparisonOperator_Is_NotEq
- core/ReplaceComparisonOperator_Is_NotLt
- core/ReplaceComparisonOperator_Is_NotLtE
- core/ReplaceComparisonOperator_Is_NotGt
- core/ReplaceComparisonOperator_Is_NotGtE
- core/ReplaceComparisonOperator_IsNot_Eq
- core/ReplaceComparisonOperator_IsNot_Lt
- core/ReplaceComparisonOperator_IsNot_LtE
- core/ReplaceComparisonOperator_IsNot_Gt
- core/ReplaceComparisonOperator_IsNot_GtE
- core/ReplaceComparisonOperator_IsNot_NotEq
- core/ReplaceComparisonOperator_IsNot_NotLt
- core/ReplaceComparisonOperator_IsNot_NotLtE
- core/ReplaceComparisonOperator_IsNot_NotGt
- core/ReplaceComparisonOperator_IsNot_NotGtE

While all of the entries in *operators-filter.exclude-operators* are treated as regular expressions, you don't need to us "fancy" regular expression features in them. As in the last two entries in the example above, you can do matching against an exact string; these are still regular expressions, albeit simple ones.

For a list of all operators in your Cosmic Ray installation, run `cosmic-ray operators`.

cr-filter-pragma

The `cr-filter-pragma` filter looks for lines in your source code containing the comment "# pragma: no mutate". Any mutation in a session that would mutate such a line is skipped.

cr-filter-git

The `cr-filter-git` filter looks for edited or new lines from the given git branch. Any mutation in a session that would mutate other lines is skipped.

By default the `master` branch is used, but you could define another one like this:

```
[cosmic-ray.filters.git-filter]
branch = "rolling"
```

External filters

Other filters are defined in separate projects.

cosmic-ray-spor-filter

The `cosmic-ray-spor-filter` filter modifies a session by skipping mutations which are indicated in a spor anchored metadata repository. In short, `spor` provides a way to associated arbitrary metadata with ranges of code, and this metadata is stored outside of the code. As your code changes, `spor` has algorithms to update the metadata (and its association with the code) automatically.

Get more details at the project page.

1.5.2 Distributors

TODO: Explain how to create a distributor.

1.5.3 Implementation

Cosmic Ray works by parsing the module under test (MUT) and its submodules into abstract syntax trees using parso. It walks the parse trees produced by parso, allowing mutation operators to modify or delete them. These modified parse trees are then turned back into code which is written to disk for use in a test run.

For each individual mutation, Cosmic Ray applies a mutation to the code on disk. It then uses user-supplied test commands to run tests against mutated code.

In effect, the mutation testing algorithm is something like this:

```
for mod in modules_under_test:
    for op in mutation_operators:
        for site in mutation_sites(op, mod):
            mutant_ast = mutate_ast(op, mod, site)
            write_to_disk(mutant_ast)

            try:
                if discover_and_run_tests():
                    print('Oh no! The mutant survived!')
                else:
                    print('The mutant was killed.')
            except Exception:
                print('The mutant was incompetent.')
```

Obviously this can result in a lot of tests, and it can take some time if your test suite is large and/or slow.

1.5.4 Mutation Operators

In Cosmic Ray we use *mutation operators* to implement the various forms of mutation that we support. For each specific kind of mutation – constant replacement, break/continue swaps, and so forth – there is an operator class that knows how to create that mutation from un-mutated code.

Implementation details

Cosmic Ray relies on parso to parse Python code into trees. Cosmic Ray operators work directly on this tree, and the results of modifying this tree are written to disk for each mutation.

Each operator is ultimately a subclass of `cosmic_ray.operators.operator.Operator`. We pass operators to various parse-tree *visitors* that let the operator view and modify the tree. When an operator reports that it can potentially modify a part of the tree, Cosmic Ray notes this and, later, asks the operator to actually perform this mutation.

Implementing an operator

To implement a new operator you need to create a subclass of `cosmic_ray.operators.operator.Operator`. The first method an operator must implement is `Operator.mutation_positions()` which tells Cosmic Ray how the operator could mutate a particular parse-tree node.

Second, an operator subclass must implement `Operator.mutate()` which actually mutates a parse-tree node.

Finally, an operator must implement the class method `Operator.examples()`. This provides a set of before and after code snippets showing how the operator works. These examples are used in the test suite and potentially for documenation purposes. An operator can choose to provide no examples simply by returning an empty iterable from `examples`, though we may decide to check for an absence of examples in the future. In any case, it's good form to provide examples.

In both cases, the operator implementation works directly with the `parso` parse tree objects.

Operator provider plugins

Cosmic Ray is designed to be extended with arbitrary operators provided by users. It dynamically discovers operators at runtime using the `stevedore` plugin system which relies on the `setuptools` `entry_points` concept.

Rather than having individual plugins for each operator, Cosmic Ray lets users specify *operator provider* plugins. An operator provider can supply any number of operators to Cosmic Ray. At a high level, Cosmic Ray finds all of the operators available to it by iterating over the operator provider plugins, and for each of those iterating over the operators that it exposes.

The operator provider API is very simple:

```
class OperatorProvider:
    def __iter__(self):
        "The sequence of operator names that this provider supplies"
        pass

    def __getitem__(self, name):
        "Get an operator class by name."
        pass
```

In other words, a provider must have a (locally) unique name for each operator it provides, it must provide an iterator over those names, and it must allow Cosmic Ray to look up operator classes by name.

To make a new operator provider available to Cosmic Ray you need to create a `cosmic_ray.operator_providers` entry point; this is generally done in `setup.py`. We'll show an example of how to do this later.

Operator naming

All operators in Cosmic Ray have a unique name for any given session. The name of an operator is based on two elements:

1. The name of the `operator_provider` entry point (i.e. as specified in `setup.py`)
2. The name that the provider associates with the operator.

The full name of an operator is simply the provider's name and the operator's name joined with "/". For example, if the provider's name was "widget_corp" and the operator's name was "add_whitespace", the full name of the operator would be "widget_corp/add_whitespace".

A full example: `NumberReplacer`

One of the operators bundled with Cosmic Ray is implemented with the clas `cosmic_ray.operators.number_replacer.NumberReplacer`. This operator looks for `Num` nodes (number literals in source code) and replaces them with new `Num` nodes that have a different numeric value. To demonstrate how to create a mutation operator and provider, we'll step through how to create that operator in a new package called `example`.

Creating the operator class

The initial layout for our package is like this:

```
setup.py
example/
  __init__.py
```

`__init__.py` is empty and `setup.py` has very minimal content:

```
from setuptools import setup

setup(
    name='example',
    version='0.1.0',
)
```

The first thing we need to do is create a new Python source file to hold our new operator. Create a file named `number_replacer.py` in the `example` directory. It has the following contents:

```
from cosmic_ray.operators.operator import Operator
import parso

class NumberReplacer(Operator):
    """An operator that modifies numeric constants."""

    def mutation_positions(self, node):
        if isinstance(node, parso.python.tree.Number):
            yield (node.start_pos, node.end_pos)
```

(continues on next page)

(continued from previous page)

```
    def mutate(self, node, index):
        """Modify the numeric value on `node`."""

        assert isinstance(node, parso.python.tree.Number)

        val = eval(node.value) + 1
        return parso.python.tree.Number(' ' + str(val), node.start_pos)
```

Let's step through this line-by-line. We first import `Operator` because we need to inherit from it:

```
from cosmic_ray.operators.operator import Operator
```

We then import `parso` because we need to use it to create mutated nodes:

```
import parso
```

We define our new operator by creating a subclass of `Operator` called `NumberReplacer`:

```
class NumberReplacer(Operator):
```

The `mutate_positions` method is called whenever Cosmic Ray needs to know if an operator can mutate a particular node. We implement ours to report a single mutation at each "number":

```
def mutation_positions(self, node):
    if isinstance(node, parso.python.tree.Number):
        yield (node.start_pos, node.end_pos)
```

Finally we implement `Operator.mutate()` which is called to actually perform the mutation. `mutate()` should return one of:

- `None` if the `node` argument should be removed from the tree, or
- a new `parso` node to replace the original one

In this case, we simply create a new `Number` node with a new value and return it:

```
def mutate(self, node, index):
    """Modify the numeric value on `node`."""

    assert isinstance(node, parso.python.tree.Number)

    val = eval(node.value) + 1
    return parso.python.tree.Number(' ' + str(val), node.start_pos)
```

That's all there is to it. This mutation operator is now ready to be applied to any code you want to test.

However, before it can really be used, you need to make it available as a plugin.

Creating the provider

In order to expose our operator to Cosmic Ray we need to create an operator provider plugin. In the case of a single operator like ours, the provider implementation is very simple. We'll put the implementation in `example/provider.py`:

```
# example/provider.py

from .number_replacer import NumberReplacer

class Provider:
    _operators = {'number-replacer': NumberReplacer}

    def __iter__(self):
        return iter(Provider._operators)

    def __getitem__(self, name):
        return Provider._operators[name]
```

Creating the plugin

In order to make your operator available to Cosmic Ray as a plugin, you need to define a new `cosmic_ray.operator_providers` entry point. This is generally done through `setup.py`, which is what we'll do here.

Modify `setup.py` with a new `entry_points` argument to `setup()`:

```
setup(
    . . .
    entry_points={
        'cosmic_ray.operator_providers': [
            'example = example.provider:Provider'
        ]
    })
```

Now when Cosmic Ray queries the `cosmic_ray.operator_providers` entry point it will see your provider - and hence your operator - along with all of the others.

1.6 Reference

1.6.1 Cosmic Ray API

cosmic_ray package

Subpackages

cosmic_ray.ast package

Submodules

cosmic_ray.ast.ast_query module

Module contents

cosmic_ray.commands package

Submodules

cosmic_ray.commands.execute module

cosmic_ray.commands.init module

cosmic_ray.commands.new_config module

Module contents

cosmic_ray.distribution package

Submodules

cosmic_ray.distribution.distributor module

Base distributor implementation details.

class cosmic_ray.distribution.distributor.**Distributor**
Bases: object

Base class for work distribution strategies.

cosmic_ray.distribution.http module

cosmic_ray.distribution.local module

Module contents

cosmic_ray.operators package

Submodules

cosmic_ray.operators.binary_operator_replacement module

cosmic_ray.operators.boolean_replacer module

cosmic_ray.operators.break_continue module

cosmic_ray.operators.comparison_operator_replacement module

cosmic_ray.operators.exception_replacer module

cosmic_ray.operators.keyword_replacer module

cosmic_ray.operators.no_op module

Implementation of the no-op operator.

class cosmic_ray.operators.no_op.**NoOp**
Bases: *cosmic_ray.operators.operator.Operator*

An operator that makes no changes.

This is primarily for baselining and debugging. It behaves like any other operator, but it makes no changes. Obviously this means that, if your test suite passes on unmutated code, it will still pass after applying this operator. Use with care.

classmethod examples()
Examples of the mutations that this operator can make.

This is primarily for testing purposes, but it could also be used for documentation.

Each example takes the following arguments: pre_mutation_code: code prior to applying the mutation. post_mutation_code: code after (successfully) applying the mutation. occurrence: the index of the occurrence to which the mutation is

applied (optional, default=0).

operator_args: a dictionary of arguments to be **-unpacked to the operator (optional, default={}).

Returns: An iterable of Examples.

mutate(*node*, *index*)
Mutate a node in an operator-specific manner.

Return the new, mutated node. Return `None` if the node has been deleted. Return `node` if there is no mutation at all for some reason.

mutation_positions(*node*)
All positions where this operator can mutate `node`.

An operator might be able to mutate a node in multiple ways, and this function should produce a position description for each of these mutations. Critically, if an operator can make multiple mutations to the same position, this should produce a position for each of these mutations (i.e. multiple identical positions).

Parameters **node** – The AST node being mutated.

Returns An iterable of `((start-line, start-col), (stop-line, stop-col))` tuples describing the locations where this operator will mutate `node`.

cosmic_ray.operators.number_replacer module

cosmic_ray.operators.operator module

Implementation of operator base class.

class cosmic_ray.operators.operator.**Argument**(*name: str*, *description: str*)
Bases: `object`

class cosmic_ray.operators.operator.**Example**(*pre_mutation_code: str*, *post_mutation_code: str*, *occurrence: Optional[int] = 0*, *operator_args: Optional[dict] = None*)
Bases: `object`

A structure to store pre and post mutation operator code snippets, including optional specification of occurrence and operator args.

This is used for testing whether the pre-mutation code is correctly mutated to the post-mutation code at the given occurrence (if specified) and for the given operator args (if specified).

occurrence = 0

operator_args = None

class cosmic_ray.operators.operator.**Operator**
Bases: abc.ABC

The mutation operator base class.

classmethod arguments() → Sequence[cosmic_ray.operators.operator.Argument]
Sequence of Arguments that the operator accepts.

Returns: A Sequence of Argument instances

classmethod examples()
Examples of the mutations that this operator can make.

This is primarily for testing purposes, but it could also be used for documentation.

Each example takes the following arguments: pre_mutation_code: code prior to applying the mutation. post_mutation_code: code after (successfully) applying the mutation. occurrence: the index of the occurrence to which the mutation is

applied (optional, default=0).

operator_args: a dictionary of arguments to be **-unpacked to the operator (optional, default={}).

Returns: An iterable of Examples.

mutate(*node*, *index*)
Mutate a node in an operator-specific manner.

Return the new, mutated node. Return `None` if the node has been deleted. Return `node` if there is no mutation at all for some reason.

mutation_positions(*node*)
All positions where this operator can mutate `node`.

An operator might be able to mutate a node in multiple ways, and this function should produce a position description for each of these mutations. Critically, if an operator can make multiple mutations to the same position, this should produce a position for each of these mutations (i.e. multiple identical positions).

Parameters **node** – The AST node being mutated.

Returns An iterable of `((start-line, start-col), (stop-line, stop-col))` tuples describing the locations where this operator will mutate `node`.

cosmic_ray.operators.provider module

cosmic_ray.operators.remove_decorator module

cosmic_ray.operators.unary_operator_replacement module

cosmic_ray.operators.util module

Utilities for implementing operators.

cosmic_ray.operators.util.**extend_name**(*suffix*)

A factory for class decorators that modify the class name by appending some text to it.

Example:

```
@extend_name('_Foo')
class Class:
    pass

assert Class.__name__ == 'Class_Foo'
```

cosmic_ray.operators.zero_iteration_for_loop module

Module contents

cosmic_ray.tools package

Subpackages

cosmic_ray.tools.filters package

Submodules

cosmic_ray.tools.filters.filter_app module

cosmic_ray.tools.filters.git module

cosmic_ray.tools.filters.operators_filter module

cosmic_ray.tools.filters.pragma_no_mutate module

Module contents

Submodules

cosmic_ray.tools.badge module

cosmic_ray.tools.html module

cosmic_ray.tools.http_workers module

cosmic_ray.tools.report module

cosmic_ray.tools.survival_rate module

cosmic_ray.tools.xml module

Module contents

Submodules

cosmic_ray.cli module

cosmic_ray.config module

cosmic_ray.exceptions module

exception cosmic_ray.exceptions.**CosmicRayTestingException**
Bases: Exception

Exception that we use for exception replacement.

cosmic_ray.modules module

Functions related to finding modules for testing.

cosmic_ray.modules.**filter_paths**(*paths*, *excluded_paths*)
Filter out path matching one of excluded_paths glob

Parameters

- **paths** – path to filter.
- **excluded_paths** – List for glob of modules to exclude.

Returns An iterable of paths Python modules (i.e. *py files).

cosmic_ray.modules.**find_modules**(*module_paths*)
Find all modules in the module (possibly package) represented by module_path.

Parameters **module_paths** – A list of pathlib.Path to Python packages or modules.

Returns An iterable of paths Python modules (i.e. *py files).

cosmic_ray.mutating module

cosmic_ray.plugins module

cosmic_ray.progress module

Management of progress reporting.

The design of this subsystem is such that progress reporting is decoupled from updating the current progress. This allows for progress reporting functions which can be invoked from another context, such as a SIGINFO signal handler.

As such, reporter callables are responsible only for displaying current progress, and must be capable of retrieving the latest progress state from elsewhere when invoked. This may be global state or reached through references that the reporter callable was constructed with. Typically the latest progress state will be updated by the routine whose progress is being monitored.

To report the current progress invoke report_progress().

To manage installation and deinstallation of progress reporting functions use the reports_progress() decorator for whole-function contexts, or the progress_reporter() context manager for narrower contexts.

It is the responsibility of the client to manage any thread-safety issues. You should assume that progress reporting functions can be called asynchronously, at any time, from the main thread.

Example:

```
_progress_message = ""

def _update_foo_progress(i, n):
    global _progress_message
    _progress_message = "{i} of {n} complete".format(i=i, n=n)

def _report_foo_progress(stream):
    print(_progress_message, file=stream)

@reports_progress(_report_foo_progress)
def foo(n):
for i in range(n):
    _update_foo_progress(i, n)

# ...

signal.signal(signal.SIGINFO,
              lambda *args: report_progress())
```

cosmic_ray.progress.**install_progress_reporter**(*reporter*)
Install a progress reporter.

Where possible prefer to use the progress_reporter() context manager or reports_progress() decorator factory.

Parameters **reporter** – A zero-argument callable to report progress. The callable provided should have the means to both retrieve and display current progress information.

cosmic_ray.progress.**progress_reporter**(*reporter*)
A context manager to install and remove a progress reporting function.

Parameters **reporter** – A zero-argument callable to report progress. The callable provided should have the means to both retrieve and display current progress information.

cosmic_ray.progress.**report_progress**(*stream=None*)
Report progress from any currently installed reporters.

Parameters **stream** – The text stream (default: sys.stderr) to which progress will be reported.

cosmic_ray.progress.**reports_progress**(*reporter*)
A decorator factory to mark functions which report progress.

Parameters **reporter** – A zero-argument callable to report progress. The callable provided should have the means to both retrieve and display current progress information.

cosmic_ray.progress.**uninstall_progress_reporter**(*reporter*)
Uninstall a progress reporter.

Where possible prefer to use the progress_reporter() context manager or reports_progress() decorator factory.

Parameters **reporter** – A callable previously installed by install_progress_reporter().

cosmic_ray.testing module

Support for running tests in a subprocess.

cosmic_ray.testing.**run_tests**(*command*, *timeout*)
Run test command in a subprocess.

If the command exits with status 0, then we assume that all tests passed. If it exits with any other code, we assume a test failed. If the call to launch the subprocess throws an exception, we consider the test 'incompetent'.

Tests which time out are considered 'killed' as well.

Parameters

- **command** (*str*) – The command to execute.
- **timeout** (*number*) – The maximum number of seconds to allow the tests to run.

Return: A tuple *(TestOutcome, output)* where the *output* is a string containing the output of the command.

cosmic_ray.timing module

Support for timing the execution of functions.

This is primarily intended to support baselining, but it's got some reasonable generic functionality.

class cosmic_ray.timing.**Timer**
Bases: object

A simple context manager for timing events.

Generally use it like this:

elapsed
Get the elapsed time between the last call to *reset* and now.

Returns a *datetime.timedelta* object.

reset()
Set the elapsed time back to 0.

cosmic_ray.version module

Cosmic Ray version info.

cosmic_ray.work_db module

cosmic_ray.work_item module

Classes for describing work and results.

class cosmic_ray.work_item.**MutationSpec**(*module_path: pathlib.Path, operator_name: str, occurrence: int, start_pos: Tuple[int, int], end_pos: Tuple[int, int], operator_args: Dict[str, Any] = <factory>*)
Bases: object

Description of a single mutation.

class cosmic_ray.work_item.**StrEnum**
Bases: str, enum.Enum

An Enum subclass with str values.

class cosmic_ray.work_item.**TestOutcome**
Bases: *cosmic_ray.work_item.StrEnum*

A enum of the possible outcomes for any mutant test run.

INCOMPETENT = 'incompetent'

KILLED = 'killed'

SURVIVED = 'survived'

class cosmic_ray.work_item.**WorkItem**(*job_id: str, mutations: Tuple[cosmic_ray.work_item.MutationSpec]*)
Bases: object

A collection (possibly empty) of mutations to perform for a single test.

This ability to perform more than one mutation for a single test run is how we support higher-order mutations.

classmethod single(*job_id*, *mutation: cosmic_ray.work_item.MutationSpec*)
Construct a WorkItem with a single mutation.

Parameters

- **job_id** – The ID of the job.
- **mutation** – The single mutation for the WorkItem.

Returns A new *WorkItem* instance.

class cosmic_ray.work_item.**WorkResult**(*worker_outcome: cosmic_ray.work_item.WorkerOutcome, output: Optional[str] = None, test_outcome: Optional[cosmic_ray.work_item.TestOutcome] = None, diff: Optional[str] = None*)
Bases: object

The result of a single mutation and test run.

diff = None

is_killed
Whether the mutation should be considered 'killed'

output = None

test_outcome = None

class cosmic_ray.work_item.**WorkerOutcome**
Bases: *cosmic_ray.work_item.StrEnum*

Possible outcomes for a worker.

ABNORMAL = 'abnormal'

EXCEPTION = 'exception'

NORMAL = 'normal'

NO_TEST = 'no-test'

SKIPPED = 'skipped'

Module contents

Cosmic Ray is a mutation testing tool for Python.

1.6.2 Commands

TODO: This is pretty wildly out of date! Perhaps we can use value-add to do this.

Details of Common Commands

Most Cosmic Ray commands use a verb-options pattern, similar to how git does things.

Possible verbs are:

- *exec*
- help
- *init*
- load
- new-config
- operators
- *dump*
- run
- worker
- apply
- baseline

Detailed information on each command can be found by running `cosmic-ray help <command>` in the terminal.

Cosmic Ray also installs a few other separate commands for producing various kinds of reports. These commands are:

- cr-report: provides a report on the status of a session
- cr-rate: prints the survival rate of a session
- cr-html: prints an HTML report on a session

Verbosity: Getting more Feedback when Running

The base command, `cosmic-ray`, has a single option: `--verbose`. The `--verbose` option changes the internal logging level from `WARN` to `INFO` and thus prints more information to the terminal.

When used with `init`, `--verbose` will list how long it took to create the mutation list and will also list which modules were found:

```
(.venv-pyerf) ~/PyErf$ cosmic-ray --verbose init --baseline=2 test_session pyerf --␣
↪pyerf/tests
INFO:root:timeout = 0.259958 seconds
INFO:root:Modules discovered: ['pyerf.tests', 'pyerf.tests.test_pyerf', 'pyerf.pyerf',
↪ 'pyerf', 'pyerf.__about__']
(.venv-pyerf) C:\dev\PyErf>cosmic-ray --verbose init --baseline=2 test_session pyerf -
↪-exclude-modules=.*tests.* -- pyerf/tests
```

(continues on next page)

(continued from previous page)

```
INFO:root:timeout = 0.239948 seconds
INFO:root:Modules discovered: ['pyerf.pyerf', 'pyerf', 'pyerf.__about__']
```

When used with `exec`, `--verbose` displays which mutation is currently being tested:

```
(.venv-pyerf) ~/PyErf$ cosmic-ray --verbose exec test_session
INFO:cosmic_ray.tasks.worker:executing: ['cosmic-ray', 'worker', 'pyerf.pyerf',
↪'number_replacer', '0', 'unittest', '--', 'pyerf/tests']
INFO:cosmic_ray.tasks.worker:executing: ['cosmic-ray', 'worker', 'pyerf.pyerf',
↪'number_replacer', '1', 'unittest', '--', 'pyerf/tests']
INFO:cosmic_ray.tasks.worker:executing: ['cosmic-ray', 'worker', 'pyerf.pyerf',
↪'number_replacer', '2', 'unittest', '--', 'pyerf/tests']
INFO:cosmic_ray.tasks.worker:executing: ['cosmic-ray', 'worker', 'pyerf.pyerf',
↪'number_replacer', '3', 'unittest', '--', 'pyerf/tests']
INFO:cosmic_ray.tasks.worker:executing: ['cosmic-ray', 'worker', 'pyerf.pyerf',
↪'number_replacer', '4', 'unittest', '--', 'pyerf/tests']
INFO:cosmic_ray.tasks.worker:executing: ['cosmic-ray', 'worker', 'pyerf.pyerf',
↪'number_replacer', '5', 'unittest', '--', 'pyerf/tests']
INFO:cosmic_ray.tasks.worker:executing: ['cosmic-ray', 'worker', 'pyerf.pyerf',
↪'number_replacer', '6', 'unittest', '--', 'pyerf/tests']
```

The `--verbose` option does not add any additional information to the `dump` verb.

Command: init

The `init` verb creates a list of mutations to apply to the source code. It has the following optional arguments:

- `--no-local-import`: Allow importing module from the current directory.

The `init` verb use following entries from the configuration file:

- `[cosmic-ray] excluded-modules = []`: Exclude modules matching those glob patterns from mutation. Use `glob.glob` syntax.

  Sample for django projects:

  ```
  excluded-modules = ["*/tests/*", "*/migrations/*"]
  ```

As mentioned in *here*, test directory can be handled via the `excluded-modules` option.

The list of files that will be mutate effectively can be show by running `cosmic-ray init` with INFO debug level:

```
cosmic-ray init -v INFO
```

Command: exec

The `exec` command is what actually runs the mutation testing.

Command: dump

The `dump` command writes a detailed JSON representation of a session to stdout.

```
$ cosmic-ray dump test_session
{"data": ["<TestReport 'test_project/tests/test_adam.py::Tests::test_bool_if' when=
↪'call' outcome='failed'>"], "test_outcome": "killed", "worker_outcome": "normal",
↪"diff": ["--- mutation diff ---", "--- a/Users/sixtynorth/projects/sixty-north/
↪cosmic-ray/test_project/adam.py", "+++ b/Users/sixtynorth/projects/sixty-north/
↪cosmic-ray/test_project/adam.py", "@@ -20,7 +20,7 @@", "     return (not object())",
↪ " ", " def bool_if():", "-    if object():", "+    if (not object()):", "
↪return True", "     raise Exception('bool_if() failed')", " "], "module": "adam",
↪"operator": "cosmic_ray.operators.boolean_replacer.AddNot", "occurrence": 0, "line_
↪number": 32, "command_line": ["cosmic-ray", "worker", "adam", "add_not", "0",
↪"pytest", "--", "-x", "tests"], "job_id": "c2bb71e6203d44f6af42a7ee35cb5df9"}
. . .
```

`dump` is designed to allow users to develop their own reports. To do this, you need a program which reads a series of JSON structures from stdin.

1.6.3 Concurrency

Note that most Cosmic Ray commands can be safely executed while `exec` is running. One exception is `init` since that will rewrite the work manifest.

For example, you can run `cr-report` on a session while that session is being executed. This will tell you what progress has been made.

1.6.4 Tests

Cosmic Ray has a number of test suites to help ensure that it works. To install the necessary dependencies for testing, run:

```
pip install -e .[dev,test]
```

`pytest` suite

The first suite is a pytest test suite that validates some if its internals. You can run that like this:

```
pytest tests/test_suite
```

The "adam" tests

There is also a set of tests which verify the various mutation operators. These tests comprise a specially prepared body of code, `adam.py`, and a full-coverage test-suite. The idea here is that Cosmic Ray should be 100% lethal against the mutants of `adam.py` or there's a problem.

We have "adam" configurations for each of the test-runner/execution-engine combinations. For example, the configuration which uses `unittest` and the `local` execution engine is in `test_project/cosmic-ray.unittest.local.conf`.

To run an "adam" test, first switch to the `test_project` directory:

```
cd tests/example_project
```

Then initialize a new session using one of the configurations. Here's an example using the `pytest/local` configuration:

```
cosmic-ray init cosmic-ray.pytest.local.conf pytest-local.sqlite
```

(Note that if you were going to use the `celery4` engine instead, you need to make sure that celery workers were running.)

Execute the session like this:

```
cosmic-ray exec pytest-local.sqlite
```

Finally, view the results of this test with `dump` and `cr-report`:

```
cr-report pytest-local.sqlite
```

You should see a 0% survival rate at the end of the report.

The full test suite

While the "adam" tests verify the various mutation operators in Cosmic Ray, the full test suite comprises a few more tests for other behaviors and functionality. To run all of these tests, it's often simplest to use tox. Just run:

```
$ tox
```

at the root of the project.

1.6.5 Continuous Integration

Cosmic Ray has a continuous integration system based on Travis. Whenever we push new changes to our github repository, travis runs a set of tests. These *tests* include low-level unit tests, end-to-end integration tests, static analysis (e.g. linting), and testing documentation builds. Generally speaking, these tests are run on all versions of Python which we support.

Automated release deployment

Cosmic Ray also has an automated release deployment scheme. Whenever you push changes to the release branch, travis attempts to make a new release. This process involves determining the release version by reading `cosmic_ray/version.py`, creating and uploading PyPI distributions, and creating new release tags in git.

Releasing a new version

As described above, the release process for Cosmic Ray is largely automatic. In order to do a new release, you simply need to:

1. Bump the version with *bumpversion*.
2. Push it to `master` on github.
3. Push the changes to the `release` branch on github.

Once the push is made to `release`, the automated release system will take over.

Note that only the Python 3.6 travis build will attempt to make a release deployment. So to see the progress of your release, check the output for that build.

1.6.6 Badge

Utility to generate badge useful to decorate your preferred Continuous Integration system (github, gitlab, ...). The badge indicate the percentage of failing migrations.

This utility is based on anybadge.

Command

```
cr-badge [--config <config_file>]  <badge_file> <session-file>
```

Configuration

```
[cosmic-ray.badge]
label = "mutation"
format = "%.2f %%"

[cosmic-ray.badge.thresholds]
50  = 'red'
70  = 'orange'
100 = 'yellow'
101 = 'green'
```

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

C

Index