Cosmic Ray Documentation Documentation

Austin Bingham

Jul 19, 2020

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 source code, running your test suite for each one. If a test suite passes on mutated code, then you have a mismatch between your tests and your functionality.

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

This documentation is a bit out of date, but we're working to get it in good order!

CHAPTER 1

Installation

1.1 `pip`

The simplest way to install Cosmic Ray is with `pip`:

```
pip install cosmic_ray
```

This will install the most recent uploaded version in PyPI.

1.2 From source

If you want to install Cosmic Ray from source you need to use `setup.py`:

```
python setup.py install
```

1.2.1 Virtual environments

You'll often want to install Cosmic Ray into its own virtual environment. In earlier versions we suggested installing Cosmic Ray into the same environment as the project being tested, but that's no longer the case. In recent versions, Cosmic Ray will create temporary virtual environments for the code under test, so the environments for Cosmic Ray and the code under test are safely separated.

CHAPTER 2

Quickstart

If you just want to get down to the business of finding and killing mutants, you still need to set a few things up.

2.1 Configurations

Before you do run any mutation tests, you need to create a *configuration file*. A configuration is 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.

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.

For example, to create a new configuration in the file `config.toml` use this command:

```
cosmic-ray new-config config.toml
```

2.1.1 Example configuration

Here's a simple example of a configuration file which uses `unittest` for testing:

```
# config.toml
[cosmic-ray]
module-path = "adam.py"
python-version = ""
timeout = 10
exclude-modules = []
test-command = "python -m unittest discover tests"

[cosmic-ray.execution-engine]
name = "local"
```

(continues on next page)

(continued from previous page)

```
[cosmic-ray.cloning]
method = "copy"
commands = []
```

You can specify a great deal of information in a configuration file, controlling things like the test execution, the execution engine, and so forth. It's entirely likely that the configuration created by `cosmic-ray new-config` won't be sufficient for your needs. Simply edit the config file to match your needs. See *Test suite* for explanations of some of those configuration options.

2.2 Create a session and run tests

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

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

Tip: You don't have to use the names `config.toml` and `my_session.sqlite`. Any names will do.

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 also create a database file called `my_session.sqlite`.

To verify that the environment is sane (that the test suite passes when it is executed by cosmic-ray), you can run the `baseline` command:

```
cosmic-ray baseline --report my_session.sqlite
```

Tip: Only one baseline can be stored in the database. If the execution failed and you fixed the environment without changing the source code, you can re-execute it with `--force` option without the need to run `init` again.

If this command succeeds, you can start executing tests with the `exec` command:

```
cosmic-ray exec my_session.sqlite
```

Unless there are errors, this won't print anything.

Tip: Because this command executes the provided test suite for every mutation it selected, it will require many times more time to execute than the whole test suite. It can be killed at any point though and restarted while keeping the status of executed mutations between the runs.

2.3 View the results

Once the execution is complete (i.e., all mutations have been performed and tested), you can see the results of your session with the `cr-report` command:

```
cr-report my_session.sqlite
```

This will print out a bunch of information about the work that was performed, including what kinds of mutants were created, which were killed, and – chillingly – which survived.

Tip: You can execute `cr-report` while `cosmic-ray exec` is running to view the progress the latter is making.

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

```
cr-html my_session.sqlite > my_session.html
```

Or use the `cr-rate` command to return error if the survival rate rose above a specified value:

```
cr-rate --fail-over 20.5 my_session.sqlite
```

Tip: `cr-rate` can also calculate confidence intervals for the survival rate when the `cosmic-ray exec` hasn't finished yet.

2.4 A concrete example: running the `adam` unittests

Cosmic Ray includes a number of unit tests which perform mutations against a simple package called `adam`. As a way of test driving Cosmic Ray, you can run these tests, too, like this:

```
cd test_project
cosmic-ray -v INFO init cosmic-ray.unittest.local.conf example-session.sqlite
cosmic-ray -v INFO exec example-session.sqlite
cr-report example-session.sqlite
```

In this case we're passing the `-v INFO` flag to the `init` and `exec` commands so that you can see what Cosmic Ray is doing. If everything goes as expected, the `cr-report` command will report a 0% survival rate.

CHAPTER 3

Theory

Mutation testing is conceptually simple and elegant. You make certain kinds of controlled changes (mutations) to your code, 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.

CHAPTER 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.

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.ReplaceBreakWithContinue`. 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`.

4.2 Execution engines

Execution engines determine the context in which tests are executed. The primary examples of execution engines are the *local* and *celery4* engines. The local engine executes tests on the local machine; the celery4 engine distributes tests to remote workers using the Celery (v4) system. Other kinds of engines might run tests on a cloud service or using other task distribution technology.

Execution engines 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 engines accomplish this.

Engines can require arbitrarily complex infrastructure and configuration. For example, the celery4 engine requires you to run rabbitmq and to attach one or more worker tasks to that queue.

Execution engines are implemented as plugins to Cosmic Ray. They are dynamically discovered, and users can create their own execution engines. Cosmic Ray includes two execution engines plugins, local and celery4.

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.

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.

4.4.1 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"
python-version = ""
timeout = 10
exclude-modules = []
test-command = python -m unittest allele_tests
execution-enging.name = "local"

[cosmic-ray.cloning]
method = 'copy'
commands = []
```

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

```
cosmic-ray init allele_config.toml allele_session.sqlite
```

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

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.

4.5.1 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 `exclude-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.

4.5.2 Executing tests

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

```
cosmic-ray exec test_session.sqlite
```

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

4.5.3 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.

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).

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
```

CHAPTER 5

Cloning

TODO: Document Cloning

When Cosmic Ray mutates code, it generally does so on a copy of the original source code. Since Cosmic Ray mutates code directly on disk, it needs to make copies of the code in order to safely run concurrent tests.

Cosmic Ray supports several methods for copying code, including simple file system copies as well as git cloning. These methods are currently hard-coded, but we'll probably provide cloning methods via plugins at some point.

You can configure cloning in your configuration TOML in the *cosmic-ray.cloning* section. At a minimum, you must have a *cosmic-ray.cloning.method* entry in your config.

The "copy" cloning method simple copies an entire filesystem directory tree. You can use configure it like this:

```
[cosmic-ray.cloning]
method = 'copy'
commands = []
```

The "git" method clones a git repository to make a clone. You can configure it like this:

```
[cosmic-ray.cloning]
method = 'git'
repo-uri = "https://github.com/project/repo.git"  # Or "." to clone the local repo
commands = [
    "pip install .[test]"
]
```

5.1 Commands

The *commands* entry is a list of shell commands that will be executed in order after the copy/clone has been created. They commands will be executed from the root of the clone and with the new virtual environment activated. You almost always need to execute at least one command to install your package, e.g. `python setup.py install`.

CHAPTER 6

Commands

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

6.1 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

6.1.1 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
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.

6.1.2 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] exclude-modules = []`: Exclude modules matching those glob patterns from mutation. Use `glob.glob` syntax.

  Sample for django projects:

  ```
  exclude-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
```

6.1.3 Command: exec

The `exec` command is what actually runs the mutation testing. There is only one optional argument: `--dist`. See *Running distributed mutation testing* for details.

6.1.4 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.

CHAPTER 7

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.

CHAPTER 8

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.

8.1 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.

8.2 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.

8.3 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.

8.3.1 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".

8.4 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`.

8.4.1 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)

    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.

8.4.2 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]
```

8.4.3 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.

CHAPTER 9

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.

Note: Filter are what used to be called *interceptors*. Interceptors were more tightly integrated with the `init` command, with many of the problems that unnecessarily high coupling often brings with it. Filters are simpler and more flexible than interceptors with no loss in power.

9.1 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.

9.1.1 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)?",
  "core/ReplaceComparisonOperator_[LG]tE_Eq",
  "core/ReplaceComparisonOperator_[LG]t_NotEq",
]
```

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

9.1.2 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.

9.1.3 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"

9.2 External filters

Other filters are defined in separate projects.

9.2.1 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.

9.3 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.

CHAPTER 10

Distributed 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 *execution engines* which can control where and how tests are run. Different engines 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.

THIS SECTION SHOULD BE CONSIDERED OUT OF DATE UNTIL FURTHER NOTICE. We will reimplement celery support, but right now it's probably broken.

10.1 The Celery execution engine

The Cosmic Ray repository includes the *celery4* execution engine. This is provided as a plugin via the *cosmic_ray_celery4_engine* package in the *plugins/execution_engines/celery3* directory. This engine uses the Celery distributed task queue to spread work across multiple nodes. (It's called "celery4" since it uses version 4 of Celery; version 4 will be available at some point as well).

The basic idea is very simple. Celery lets you start multiple *workers* which listen for commands from a task queue. A central process creates all of the commands for a mutation testing run, and these commands are distributed to the workers as they become available. When a worker receives a command, it starts a *new* python process (using the `worker` subcommand to Cosmic Ray) which performs a single mutation and runs the test suite.

Spawning a separate process for each test suite may seem expensive. However, it's the best way we have for ensuring that pathological mutants can't somehow corrupt the runtime of the worker processes. And ultimately the cost of starting the process is likely to be very small compared to the runtime of the test suite.

By its nature, Celery lets you start workers on as many systems as you want, all connected to the same task queue. So you could potentially have thousands of workers performing mutation testing runs, giving nearly perfect scaling! While not everyone has thousands of machines on hand to do their testing work, it's conceivable that Cosmic Ray will

one day be able to work with machines on commodity cloud providers, meaning that highly-scaled mutation testing for Python will be available to anyone who wants it.

10.1.1 Installing the *celery4* worker

The *cosmic_ray_celery34engine* package is installed separately from *cosmic_ray* itself. This is primarily so that *cosmic_ray* doesn't have a direct dependency on any version of *celery*.

To install the plugin, you need to run this command from *plugins/execution-engines/celery4*:

```
python setup.py install
```

This will install and register the plugin.

10.1.2 Installing RabbitMQ

Celery is primarily a Python API atop the RabbitMQ task queue. As such, if you want to use Cosmic Ray in distributed mode you first need to install RabbitMQ and run the server. The steps for installing and running RabbitMQ are covered in detail at that project's site, so go there for more information. Make sure the RabbitMQ server is installated and running before going any further with distributed execution.

10.1.3 Starting distributed worker processes

Once RabbitMQ is running, you need to start some worker processes which will do the actually mutation testing. Start one or more worker processes like this:

```
celery -A cosmic_ray_celery4_engine.worker worker
```

You should do this, of course, from the virtual environment into which you've installed Cosmic Ray. Similarly, you need to make sure that the worker is in an environment in which it can import the modules under test. Generally speaking, you can meet both of these criteria if you install Cosmic Ray into and run workers from a virtual environment into which you've installed the modules under test.

10.1.4 Running distributed mutation testing

Aside from starting workers, you also need to specify *celery4* in your configuration. For example, instead of a "local" configuration like this:

```
execution-engine.name = "local"
```

You would use the name "celery4" like this:

```
execution-engine.name = "celery4"
```

With this configuration in place, you then need to do an *init* to create a session followed by *exec* to run the tests:

```
cosmic-ray init my_config my_session
cosmic-ray exec my_session
```

This *exec* will distribute testing runs to your celery workers.

CHAPTER 11

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.

CHAPTER 12

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]
```

12.1 `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
```

12.2 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.

12.3 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.

CHAPTER 13

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.

13.1 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.

13.1.1 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.

CHAPTER 14

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.

14.1 Command

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

14.2 Configuration

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

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

CHAPTER 15

Indices and tables

- genindex
- modindex
- search