

---

## Cane

Fails your build if code quality thresholds are not met.

Discipline will set you free.

**Cane still technically works, but for new projects you're probably better off using Rubocop and customizing it to your liking. It is far more comprehensive and more widely used.**

## Usage

```
1 gem install cane
2 cane --abc-glob '{lib,spec}/**/*.rb' --abc-max 15
```

Your main build task should run this, probably via `bundle exec`. It will have a non-zero exit code if any quality checks fail. Also, a report:

```
1 > cane
2
3 Methods exceeded maximum allowed ABC complexity (2):
4
5   lib/cane.rb  Cane#sample      23
6   lib/cane.rb  Cane#sample_2  17
7
8 Lines violated style requirements (2):
9
10  lib/cane.rb:20  Line length >80
11  lib/cane.rb:42  Trailing whitespace
12
13 Class definitions require explanatory comments on preceding line (1):
14  lib/cane:3  SomeClass
```

Customize behaviour with a wealth of options:

```
1 > cane --help
2 Usage: cane [options]
3
4 Default options are loaded from a .cane file in the current directory.
5
6 -r, --require FILE           Load a Ruby file containing user-
   defined checks
7 -c, --check CLASS           Use the given user-defined check
8
9   --abc-glob GLOB           Glob to run ABC metrics over (default:
   {app,lib}/**/*.rb)
10  --abc-max VALUE           Ignore methods under this complexity (
   default: 15)
```

---

```

11  --abc-exclude METHOD      Exclude method from analysis (eg. Foo
    : :Bar#method)
12  --no-abc                 Disable ABC checking
13
14  --style-glob GLOB        Glob to run style checks over (default
    : {app,lib,spec}/**/*.rb)
15  --style-measure VALUE    Max line length (default: 80)
16  --style-exclude GLOB     Exclude file or glob from style
    checking
17  --no-style               Disable style checking
18
19  --doc-glob GLOB          Glob to run doc checks over (default:
    {app,lib}/**/*.rb)
20  --doc-exclude GLOB       Exclude file or glob from
    documentation checking
21  --no-readme              Disable readme checking
22  --no-doc                 Disable documentation checking
23
24  --lt FILE,THRESHOLD      Check the number in FILE is < to
    THRESHOLD (a number or another file name)
25  --lte FILE,THRESHOLD    Check the number in FILE is <= to
    THRESHOLD (a number or another file name)
26  --eq FILE,THRESHOLD     Check the number in FILE is == to
    THRESHOLD (a number or another file name)
27  --gte FILE,THRESHOLD    Check the number in FILE is >= to
    THRESHOLD (a number or another file name)
28  --gt FILE,THRESHOLD     Check the number in FILE is > to
    THRESHOLD (a number or another file name)
29
30  -f, --all FILE           Apply all checks to given file
31  --max-violations VALUE   Max allowed violations (default: 0)
32  --json                  Output as JSON
33  --parallel              Use all processors. Slower on small
    projects, faster on large.
34  --color                 Colorize output
35
36  -v, --version            Show version
37  -h, --help              Show this message

```

Set default options using a `.cane` file:

```

1  > cat .cane
2  --no-doc
3  --abc-glob **/*.rb
4  > cane

```

It works exactly the same as specifying the options on the command-line. Command-line arguments will override arguments specified in the `.cane` file.

---

## Integrating with Rake

```
1 begin
2   require 'cane/rake_task'
3
4   desc "Run cane to check quality metrics"
5   Cane::RakeTask.new(:quality) do |cane|
6     cane.abc_max = 10
7     cane.add_threshold 'coverage/covered_percent', :>=, 99
8     cane.no_style = true
9     cane.abc_exclude = %w(Foo::Bar#some_method)
10  end
11
12  task :default => :quality
13  rescue LoadError
14    warn "cane not available, quality task not provided."
15  end
```

Loading options from a `.cane` file is supported by setting `cane_file=` to the file name.

Rescuing `LoadError` is a good idea, since `rake -T` failing is totally frustrating.

## Adding to a legacy project

Cane can be configured to still pass in the presence of a set number of violations using the `--max-violations` option. This is ideal for retrofitting on to an existing application that may already have many violations. By setting the maximum to the current number, no immediate changes will be required to your existing code base, but you will be protected from things getting worse.

You may also consider beginning with high thresholds and ratcheting them down over time, or defining exclusions for specific troublesome violations (not recommended).

## Integrating with SimpleCov

Any value in a file can be used as a threshold:

```
1 > echo "89" > coverage/.last_run.json
2 > cane --gte 'coverage/.last_run.json,90'
3
4 Quality threshold crossed
5
6   coverage/covered_percent is 89, should be >= 90
```

---

## Implementing your own checks

Checks must implement:

- A class level `options` method that returns a hash of available options. This will be included in help output if the check is added before `--help`. If your check does not require any configuration, return an empty hash.
- A one argument constructor, into which will be passed the options specified for your check.
- A `violations` method that returns an array of violations.

See existing checks for guidance. Create your check in a new file:

```
1 # unhappy.rb
2 class UnhappyCheck < Struct.new(:opts)
3   def self.options
4     {
5       unhappy_file: ["File to check", default: [nil]]
6     }
7   end
8
9   def violations
10    [
11      description: "Files are unhappy",
12      file:       opts.fetch(:unhappy_file),
13      label:      ":(("
14    ]
15  end
16 end
```

Include your check either using command-line options:

```
1 cane -r unhappy.rb --check UnhappyCheck --unhappy-file myfile
```

Or in your rake task:

```
1 require 'unhappy'
2
3 Cane::RakeTask.new(:quality) do |c|
4   c.use UnhappyCheck, unhappy_file: 'myfile'
5 end
```

## Protips

### Writing class level documentation

Classes are commonly the first entry point into a code base, often for an oncall engineer responding to an exception, so provide enough information to orient first-time readers.

---

A good class level comment should answer the following:

- Why does this class exist?
- How does it fit in to the larger system?
- Explanation of any domain-specific terms.

If you have specific documentation elsewhere (say, in the README or a wiki), a link to that suffices.

If the class is a known entry point, such as a regular background job that can potentially fail, then also provide enough context that it can be efficiently dealt with. In the background job case:

- Should it be retried?
- What if it failed 5 days ago and we're only looking at it now?
- Who cares that this job failed?

## **Writing a readme**

A good README should include at a minimum:

- Why the project exists.
- How to get started with development.
- How to deploy the project (if applicable).
- Status of the project (spike, active development, stable in production).
- Compatibility notes (1.8, 1.9, JRuby).
- Any interesting technical or architectural decisions made on the project (this could be as simple as a link to an external design document).

## **Compatibility**

Requires CRuby 2.0, since it depends on the [ripper](#) library to calculate complexity metrics. This only applies to the Ruby used to run Cane, not the project it is being run against. In other words, you can run Cane against your 1.8 or JRuby project.

## **Support**

Make a new github issue.

---

## **Contributing**

Fork and patch! Before any changes are merged to master, we need you to sign an Individual Contributor Agreement (Google Form).