coala Documentation

Release 0.11.0

The coala Developers

Nov 02, 2018

Home

CHAPTER 1

Installing coala Natively

Note: If you have a working docker and you do not want to work with python, go to the bottom on the docker setup.

This page will run you through the installation of coala without docker. coala currently supports Linux, Windows and OS X.

1.1 Installing Python and Pip

In order to use coala, you need Python installed. In order to do so, you should install Python >= 3.4 from here.

The easiest way to install coala is using pip (Pip Installs Packages). If you don't already have pip, you can install it like described in the pip installation guide.

Note: Pip is shipped with recent Python versions by default.

To check whether you have pip installed, type the following command which will also show you more information about your current pip version:

```
$ pip show pip
```

Make sure you have pip >= 8 installed as older versions might prevent coala from being installed properly.

1.2 Installing coala

There are three ways of installing coala. By using a virtualenv, by installing it system-wide or directly from source.

After successfully installing coala, you will need to install all the dependencies the bears have.

Note: If Java is installed after coala, you will need to restart your shell.

1.2.1 System Wide Installation

The simplest way to install coala is to do it system-wide, but this is generally discouraged in favor of using a virtualenv.

To install the latest most stable version of coala and supported bears system-wide, use:

```
$ pip3 install coala-bears
```

Note: For this and all future steps, some steps require root access (also known as administrative privileges in Windows).

Unix based (OS X, Linux) - This can be achieved by using `sudo` in front of the command, as in: `sudo command_name` instead of `command_name`

Windows - The easiest way on Windows is to start a command prompt as an administrator and start `setup.py`.

To install the nightly build from our master branch, you can do:

```
$ pip3 install coala-bears --pre
```

To install coala only (without any bears), you can do:

```
$ pip3 install coala
```

Note: **Ubuntu Users** - Do not use `apt-get install coala` for installing coala as that is different software.

1.2.2 Installing inside a pipenv

Pipenv is a user-friendly method of managing virtual environments and packages. You can look at it as a mix of `pip` and `virtualenv`, so you don't have to manage them seperately for your projects.

We'll now install `pipenv` by running the following command:

```
$ pip3 install pipenv
```

Now we create a Python 3 environment for the project. Move to your project directory and run the following command:

```
$ pipenv --three
```

The above command will create a virtual environment isolated from your system global installation.

To install coala and all bears, run the following command:

```
$ pipenv install coala-bears
```

Great, you have now setup a pipenv environment for your project directory and installed coala using it. You can now activate the pipenv by running:

```
$ pipenv shell
```

Your virtual environment (virtualenv) is now active. You'll notice the change before the `$` sign on the command-line which will have your project directory name and some alpha-numeric value to make it unique. You can use coala by running the `coala` command and exit the virtualenv environment by entering `exit` at any time.

You can read more about pipenv in its documentation.

1.2.3 Installing inside a virtualenv

Although `pipenv` sorts all your package and virtual environment needs, if you still feel the need to use `virtualenv` separately you can follow the steps given below.

You can read more about it at the virtualenv documentation.

First, we need to install virtualenv to the system. You may already have this installed as `virtualenv` or `pyvenv`. If you do not, this can be done with `pip3` easily:

```
$ pip3 install virtualenv
```

Once you have virtualenv installed, just fire up a shell and create your own environment. I usually create a project folder and a `venv` folder:

```
$ virtualenv venv
```

Note: If you have both Python 3 and Python 2 installed, use `virtualenv venv -p $(which python3)`. This creates an isolated Python 3 environment named `venv` in your current directory, as coala only works for Python 3.4 and above.

Now, whenever you want to work on the project, you only have to activate the corresponding environment.

On **Unix based** systems (OS X and Linux), this can be done with:

```
$ source venv/bin/activate
```

And on **Windows** this is done with:

```
$ venv\scripts\activate
```

Finally, you should install coala and the supported bears inside the activated virtualenv with:

```
(venv)$ pip3 install coala-bears
```

CHAPTER 2

Using coala With Docker

Use this installation method if you *just simply want to use coala*.

The recommended way to use coala is using docker: coala has a lot of dependencies because it has so much code analysis for so many languages. If you use our docker image, you can run it like any other tool but you do not need to care about those! The general command to run coala is:

```
$ docker run -ti -v $(pwd):/app --workdir=/app coala/base coala --version
```

Note: This will automatically download the docker image with all the coala dependencies for you. The image may take up a bit over 2GB of space on your disk. Check out the native installation if this is not for you.

2.1 Windows

If you use Windows, you can install docker easily using Docker Toolbox.

2.1.1 Bleeding edge installation

If you would like to develop coala, you should check out our Newcomer Tutorial and *get in touch with us*.

```
$ pip3 install coala-bears --pre
```

Also check out the Developers Setup docs.

2.1.2 Alternate location installation

If you want to install coala to an alternate location, you can e.g. call `python3 setup.py install --prefix=/your/prefix/location`. Other installation options are documented in the Python docs.

Note: If you are using a proxy, follow these steps:

- Set up your system-wide proxy.
- Use `sudo -E pip3 install coala` (the `-E` flag takes the existing environment variables into the `sudo` environment).

You could also set your pip.conf file to use a proxy. To find out more, read Using pip behind a proxy on StackOverflow for further clarification.

2.2 Optional Dependencies

Those dependencies are not mandatory. You may install all of the dependencies if you want to install all the bears. The bear application also asks for the packages needed in case it does not have it.

The requirements files (Gemfile, requirements.txt, etc.) are in the coala-bears repository and you should not get them from source, but you should git clone the repository if you want to execute those commands.

This section lists dependencies of coala that are not automatically installed. On Windows, you can get many with `nuget` (https://www.nuget.org/). On Mac, Homebrew will help you installing dependencies (http://brew.sh/). These dependencies require you to have the repository cloned locally.

2.2.1 JS Dependencies

coala features a lot of bears that use linters written in JavaScript. In order for them to be usable, you need to install them via `npm` (http://nodejs.org/), while in the project directory:

```
$ npm install -g
```

If a bear still doesn't work for you, please make sure that you have a recent version of `npm` installed. Many linux distributions ship a very old one.

2.2.2 Ruby Dependencies

There are also a few bears which rely on Ruby Gems. In order to install them, you will need `Gem` (https://rubygems.org/pages/download/) installed and `bundler`.

To grab `bundler`, use:

```
$ gem install bundler
```

Then, simply run:

```
$ bundle install
$ git add Gemfile Gemfile.lock
```

2.2.3 Binary Dependencies

Some bears need some binary dependencies. Some of those include:

- PHPLintBear: Install `php`

- GNUIndentBear: Install `indent` (be sure to use GNU Indent, Mac ships a non-GNU version that lacks some functionality.)
- CSharpLintBear: Install `mono-mcs`

For further help with installing bears with binary dependencies, don't hesitate to *get in touch with us*.

2.2.4 Clang

coala features some bears that make use of Clang. In order for them to work, you need to install `libclang`:

- Ubuntu: `apt-get install libclang1`
- Fedora: `dnf install clang-libs` (Use `yum` instead of `dnf` on Fedora 21 or lower.)
- ArchLinux: `pacman -Sy clang`
- Windows: `nuget install ClangSharp`
- OS X: `brew install llvm --with-clang`

If these do not help you, search for a package that contains `libclang.so`.

On Windows, you need to execute this command to add the libclang path to the *PATH* variable permanently (you need to be an administrator):

```
setx PATH "%PATH%;%cd%\ClangSharp.XXX\content\x86" \M
```

For x86 python or for x64 python:

```
setx PATH "%PATH%;%cd%\ClangSharp.XXX\content\x64" \M
```

Replace "XXX" with the ClangSharp version you received from nuget.

2.3 Shell-Autocompletion Support

If you are a bash/zsh user , checkout the guide to set up autocompletion for coala arguments and bear names.

2.4 Installation Errors

In case you are getting `ValueError:('Expected version spec in', 'appdirs ~=1.4.0', 'at', ' ~=1.4.0')`, then don't panic. It happens when you are using an outdated version of pip that doesn't support our version specifiers yet.

Ideally, you have to create a virtual environment with a newer pip:

```
$ pip3 install virtualenv
$ virtualenv -p python3 ~/venv/coala
$ . ~/venv/coala/bin/activate
$ pip install -U pip
$ pip install coala-bears
```

You have to activate this virtualenv on every terminal session you want to use coala though (tip: add it to bashrc!).

2.4.1 Generating Documentation

coala documentation is in a separate repository. First you need to install the requirements:

```
$ pip3 install -r requirements.txt
```

To generate the documentation coala uses *sphinx*. Documentation can be generated by running the following command while in root directory of the repository:

```
$ make html
```

You can then open `_build\html\index.html` in your favourite browser.

See Writing Documentation for more information.

CHAPTER 3

Getting Started with coala

Welcome to this little tutorial. It is meant to be a gentle introduction to the usage of coala.

3.1 Prerequisites

In order to complete this tutorial you will need coala installed. Installation instructions can be found here.

Note: Here's a list of our supported languages.

3.2 Get Some Code

In order to perform a static code analysis on your code you will need some code to check. If you do not have your own code you want to check, you can retrieve our tutorial samples:

```
git clone https://github.com/coala/coala-tutorial
```

Please note that the commands given in this tutorial are intended for use with this sample code and may need minor adjustments.

3.3 Let's Start!

There are two options how to let coala know what kind of analysis it should perform on which code.

3.3.1 Command Line Interface

In order to specify the files to analyze, you can use the `--files` argument of coala like demonstrated below. For all file paths, you can specify (recursive) globs.

Because analysis routines can do many various things we named them **bears**. A bear can check your code for potential problems, calculate metrics and even provide corrections for your code.

You can specify the bears that you want coala to run using the `--bears` argument:

```
cd coala-tutorial
coala --files=src/\*.c --bears=SpaceConsistencyBear --save
```

Note: You can use comma separated values to specify more than one item in arguments! Do not use spaces as that would start a new argument. Example: `SpaceConsistencyBear,PEP8Bear`

coala will now ask you for missing values that are needed to perform the analysis, which in this case is only the `use_spaces` setting. We recommend setting it to `True`.

```
Please enter a value for the setting "use_spaces" (True if spaces
are to be used instead of tabs.) needed by SpaceConsistencyBear
for section "cli"
```

coala will now check the code and, in case you use the tutorial code, yield one result. SpaceConsistencyBear will detect a trailing whitespace at the end of the line, after `#include <stdio.h>` in the `main.c` file. coala will then ask you to remove the trailing space, by applying the suggested patch (option 2).

```
Executing section cli...

src/main.c
|   1| #include·<stdio.h>·
|    | [NORMAL] SpaceConsistencyBear:
|    | Line contains following spacing inconsistencies:
|    | - Trailing whitespaces.
|----|    | /path/coala-tutorial/src/main.c
|    |++++| /path/coala-tutorial/src/main.c
|   1|    |-#include <stdio.h>
|    |   1|+#include <stdio.h>
|   2|   2|
|   3|   3| int main(void) {
|   4|   4|     printf("Welcome to coala. Keep following the
                tutorial, you are doing a great job so far!\n");
|    | *0: Do nothing
|    |  1: Open file(s)
|    |  2: Apply patch
|    |  3: Print more info
|    |  4: Add ignore comment
|    | Enter number (Ctrl-D to exit): 2
```

If the patch was applied succesfully, you should see something like this:

```
|    | Patch applied successfully.
|    | *0: Do nothing
|    |  1: Open file(s)
|    |  2: Print more info
|    |  3: Add ignore comment
|    | Enter number (Ctrl-D to exit):
```

Exit by pressing Ctrl-D.

You can also run coala in non interactive mode (given that all the settings required by the bears you are using are provided in the `.coafile`)

```
coala --non-interactive
```

In this case there won't be any interaction, the patch will be shown directly.

Feel free to experiment a bit. You've successfully analysed some code! But don't stop reading - you don't have to enter all those values again! We have given coala the `--save` argument, which means that it will automatically generate a `.coafile` into the current directory. Read on!

3.3.2 Configuration Files - coafiles

coala supports a very simple configuration file. If you've executed the instructions from the CLI section above, coala will already have such a file readily prepared for you. Go, take a look at it:

```
cat .coafile
```

Note: If you are using Windows, you should use `type .coafile` instead!

This should yield something like this:

```
[cli]
bears = SpaceConsistencyBear
files = src/*.c
use_spaces = True
```

If you now invoke `coala` it will parse this `.coafile` from your current directory. This makes it easy to specify once for your project what is checked with which bears and make it available to all contributors.

Feel free to play around with this file. You can either edit it manually or add/edit settings via `coala --save ...` invocations. If you want coala to save settings every time, you can add `save = True` manually into your `.coafile`.

3.4 Sections

Thats all nice and well but we also have a Makefile for our project we want to check. So let us introduce another feature of our configuration syntax: *sections*.

The line `[cli]` in the `.coafile` implies that everything below belongs to the special `cli` section. You may specify sections when you enter the settings via the Command Line Interface (CLI). You will soon learn all about them. When you don't specify any sections, the settings will implicitly belong to the `[cli]` section.

Next you will see how to specify sections using the command line when you are running coala. Let's check the line lengths of our Makefile:

```
coala -S Makefiles.bears=LineLengthBear Makefiles.files=Makefile --save
```

As you can see, the `-S` (or `--settings`) option allows to specify arbitrary settings. Settings can be directly stored into a section with the `section.setting` syntax.

By default, the `LineLengthBear` checks whether each line contains `79` chars or less in a line. To change this value, use the `max_line_length` inside the `.coafile`.

coala will now yield any result you didn't correct last time, plus a new one for the Makefile. This time coala (or better, the `LineLengthBear`) doesn't know how to fix the issue but still tries to provide as much helpful information as possible and provides you the option to directly open the file in an editor of your choice.

Note: If you want to set a default editor and not be asked for one every time, you can simply add `editor=$editorName` (i.e. editor=vim) to your project's `.coafile` and it will automatically open in that one.

Note: If your editor is already open this may not work, because the other process will shortly communicate with the existent process and return immediately. coala handles this for some editors automatically, if yours does not work yet - please file an issue so we can include it!

If you changed one file in multiple results, coala will merge the changes if this is possible.

coala should have appended something like this to your `.coafile`:

```
[Makefiles]
bears = LineLengthBear
files = Makefile
```

As you see, sections provide a way to have different configurations for possibly different languages in one file. They are executed sequentially.

Note: For a list of configuration options for the bears, take a look at our coala languages directory.

3.5 Auto-applying Results

Often you don't want to look at trivial results like spacing issues. For that purpose coala includes a special setting called `default_actions` that allows you to set the action for a bear that shall be automatically applied on run. We have a command line alias `--apply-patches` to make it easier to use.

By using `--apply-patches`, the user does not have to press `2. (A)pply patch` for applying a patch. Every patch is applied automatically. Alternatively, using the setting `default_actions="*: ApplyPatchAction"` will automatically apply `--apply-patches` on run.

Let's automatically fix Python code. Take a look at our sample Python code:

```
$ cat src/add.py

"""
This is a simple library that provides a function that can add numbers.

Cheers!
"""



def add(a,b):
    return a+b;
```

```
import sys
```

That looks horrible, doesn't it? Let's fix it!

```
$ coala -S python.bears=PEP8Bear python.files=\*\*/\*.py \
--apply-patches --save
# other output ...
Executing section cli...
Executing section python...
[INFO][11:03:37] Applied 'ApplyPatchAction' for 'PEP8Bear'.
[INFO][11:03:37] Applied 'ApplyPatchAction' for 'PEP8Bear'.
```

coala would now fix all spacing issues and without bothering you again.

Note: When you try the above example, you may get a warning, saying that all settings in the `cli` section are implicitly inherited to all other sections (if they do not override their values). It also advises us to change the name of that section to avoid unexpected behavior. The next section explains what it means and how you can avoid it.

3.6 Setting Inheritance

Let's first see what inheritance means.

Before proceeding, rename the `cli` section in the `.coafile` to `all` (we will soon explain the reason behind this change).

Lets add the following section to our `.coafile`:

```
[all.TODOS]
bears = KeywordBear
```

And execute `coala` with the `-s` argument which is the same as `--save`. I recommend setting case insensitive keywords to `TODO, FIXME` and case sensitive keywords empty.

After the results we've already seen, we'll see a new informational one which informs us that we have a TODO in our code.

Did you note that we didn't specify which files to check this time? This is because all settings, including `files = src/*.c`, from the `all` section (previously called `cli`) have been inherited in the new `TODOS` section that we just added.

You can make a section inherit from any previously defined section using this syntax:

```
[parentSection.childSection]
```

Note: `cli` is an internally reserved section name. All of its settings are implicitly inherited to every other section by default. It is because of this implicit inheritance feature that we are adviced to rename the `cli` section to something else. Doing so will save us from having unexpected values of `cli` being implicitly inherited into our sections. We strongly suggest renaming it.

3.7 Ignoring Issues

There are several ways to ignore certain issues, so you aren't lost if any routines yield false positives.

3.7.1 Ignoring Files

coala lets you ignore whole files through the `ignore` setting. In addition to normal globs, coala offers `**` to match all directories and subdirectories:

```
files = **/*.h
ignore = **/resources.h
```

This configuration would include all header (`.h`) files but leaves out resource headers.

3.7.2 Ignoring Code Inside Files

Sometimes you need finer-graded ignores. Imagine you have a `LineLengthBear` that shall not run on some code segments, because you can't wrap them:

```
code = "that's checked normally"

# Ignore LineLengthBear
unwrappable_string = "some string that is long and would exceed the limit"
```

You can also skip an area:

```
# Start ignoring LineLengthBear
unwrappable_string_2 = unwrappable_string + "yeah it goes even further..."
another_unwrappable_string = unwrappable_string + unwrappable_string_2
# Stop ignoring
```

You can also conditionally combine ignore rules! Bear names will be split by comma and spaces, invalid bear names like `and` will be ignored.

Also note that in the bear names delimited by commas and spaces, you may specify glob wildcards that match several bears:

```
# Start ignoring Line*, Py*
unwrappable_string_2 = unwrappable_string + "yeah it goes even further..."
another_unwrappable_string = unwrappable_string + unwrappable_string_2
# Stop ignoring
```

In the above example all bears matching the glob *Line** and *Py** will be ignored. You may also specify more complex globs here such as *# Start ignoring (Line*|P[yx]*)* which will ignore all bears' names which start with *Line*, *Py*, and *Px*.

```
# Ignore LineLengthBear and SpaceConsistencyBear
    variable = "Why the heck are spaces used instead of tabs..." + "so_long"
```

If you put an `all` instead of the bear names directly after the `ignore`/`ignoring` keyword, the results of all bears affecting those lines will be ignored.

If you've used another linter in the past, you don't have to change your pre-existing code with the `noqa` keywords to `ignore` as the examples below work as well. If no bears are specified, `noqa` will be applicable to work for all bears.

```
# noqa
long_line = "This is a long line ... "
```

If you wish to specify which bear to use with `noqa`, as is done with `ignore`, you would have to proceed as follows:

```
# noqa LineLengthBear
long_line = "This is a long line ... "
```

3.8 Enabling/Disabling Sections

Now that we have sections we need some way to control, which sections are executed. coala provides two ways to do that:

3.8.1 Manual Enabling/Disabling

If you add the line `TODOS.enabled=False` to some arbitrary place to your `.coafile` or just `enabled=False` into the `TODOS` section, coala will not show the TODOs on every run.

Especially for those bears yielding informational messages which you might want to see from time to time this is a good way to silence them.

3.8.2 Specifying Targets

If you provide positional arguments, like `coala Makefiles`, coala will execute exclusively those sections that are specified. This will not get stored in your `.coafile` and will take precedence over all enabled settings. You can specify several targets separated by a space.

What was that TODO again?

3.9 Continuing the Journey

If you want to know about more options, take a look at our coala settings documentation or with `coala -h`. If you liked or disliked this tutorial, feel free to drop us a note at our bug tracker or mailing list.

If you need more flexibility, know that coala is extensible in many ways due to its modular design:

- If you want to write your own bears, take a look at our tutorial.
- If you want to add custom actions for results, take a look at the code in coalib/results/results_actions.
- If you want to have some custom outputs (e.g. HTML pages, a GUI or voice interaction) take a look at modules lying in coalib/output.

Happy coding!

CHAPTER 4

Writing a coala Configuration File (coafile and coarc)

This document gives a short introduction to the specification of a coala configuration file. It is meant to be rather factual. If you wish to learn by example, please take a look at *Getting Started with coala*. It also teaches how to change settings inside a coala file to suit your taste.

4.1 Naming, Scope and Location

You can use up to three coafiles to configure your project.

1. A project-wide coafile.
2. A user-wide coafile.
3. A system-wide coafile.

4.1.1 Project-Wide coafile

It is a convention that the project-wide coafile is named `.coafile` and lies in the project root directory. If you follow this convention, simply executing `coala` from the project root will execute the configuration specified in that file.

Settings given in the project-wide coafile override all settings given by other files and can only be overridden by settings given via the command line interface.

4.1.2 User-Wide and System-Wide coafile

You can place a `.coarc` file in your home directory to set certain user wide settings. Those settings will automatically be taken for all projects executed with that user.

All settings specified here override only settings given by the system wide coafile which has the lowest priority. The `system_coafile` must lie in the coala installation directory and is valid for everyone using this coala installation.

It can be used to define the type of files you usually don't want to lint, like minified files (e.g. `*.min.js`) and backup files (e.g. `*.orig`):

```
ignore = **.min.js, **.orig
```

4.2 Explicit Setting Inheritance

Every coafile contains one or more sections. Section names are case insensitive. The old(pre 0.11.x) implicit section inheritance syntax has been deprecated and has been scheduled for removal in coala version 0.12.0. Instead, define section inheritance explicitly by naming a section in the format `[basesection.newsection]`. Extra values can be appended to an inherited setting using the += operator.

Consider the following coafile:

```
[all]
enabled = True
overridable = 2
ignore = vendor1/

[all.section1]
overridable = 3
ignore += vendor2/
other = some_value

[all.section2]
overridable = 4
ignore += vendor3/
other = some_other_value
```

This is the same file without section inheritance:

```
[all]
enabled = True
overridable = 2
ignore = vendor1/

[section1]
enabled = True
overridable = 3
ignore = vendor1/, vendor2/
other = some_value

[section2]
enabled = True
overridable = 4
ignore = vendor1/, vendor3/
other = some_other_value
```

All settings must be part of a section, so don't do this for implicit inheritance (this is also deprecated behavior). Implicit inheritance was leading to a section automatically getting inherited to all other sections without semantically making sense:

```
# bad!
setting1 = 1

[section1]
# setting1 is inherited
setting2 = 2
```

Instead, make the inheritance explicit:

```
# better!
[all]
setting1 = 1

[all.section1]
# setting1 is inherited
setting2 = 2
```

4.3 Defining Aspects and Tastes

Aspects is an alternative way to configure coala. In this mode, we don't need to explicitly state list of bears, coala will choose it automatically based on requested aspects in coafile. To run coala in this mode, we need to define *aspects*, *files*, *languages*, and optionally aspect tastes setting. See the following example:

```
[all]
files = **
aspects = aspectname1, AspectName2 # case-insensitive
# defining an aspect's taste
aspectname1:aspect_taste = 80
# we can define subaspect taste through its parent
aspectname1:subaspect_taste = word1, word2, word3

[all.python]
files = **.py
language = Python
# appending additional aspect
aspects += aspectname3
# excluding certain subaspect
excludes = AspectName2Subaspect
```

4.3.1 Comments, Escaping and Multiline Values and Keys

Comments are simply done with a preceding `#`. If you want to use a `#` within a value, you can simply escape it:

```
a_key = a\#value # And a comment at the end!
```

Any line not containing an unescaped = is simply appended to the value of the last key:

```
a_key = a
value
# this is not part of the value
that /= is
very long!
```

Similarly, you can also set a value to multiple keys: `key_1, key_2 = value` is equivalent to `key_1 = value` and `key_2 = value` in separate lines.

As the backslash `\` is the escape character it is recommended to use forward slashes `/` as path separator even on Windows (to keep relative paths platform independent), use double-backslashes `\\` if you really mean a backslash in all places.

You can now proceed to an example with *Getting Started with coala*.

CHAPTER 5

Using coala-quickstart to Generate a coala Configuration File

This document aims to make people aware of coala-quickstart by providing a brief introduction about its features and how to quickly generate coala configuration files for your projects.

5.1 What is coala-quickstart?

coala-quickstart is a CLI tool that helps users to quickly get started with coala by generating a `.coafile` tailored for their project. The `.coafile` is generated based on the questions answered by the users about their project.

5.2 Features

coala-quickstart offers the following features:

- Out-of-the-box support for projects using various popular languages such as C/C++, Python, JavaScript and many more with built-in check routines.
- Automatic detection of languages used in your project.
- Automatic identification of bears that might be relevant for your project and detection of bear settings based on the languages used.
- A clean and simple interface with a well defined flow.

5.3 Installation

To install the latest stable version run:

```
$ pip3 install coala-quickstart
```

To install the latest development version run:

```
$ git clone https://github.com/coala/coala-quickstart.git
$ cd coala-quickstart
$ pip3 install .
```

5.4 Usage

To get started simply run:

```
$ coala-quickstart
```

This should prompt you for your project's directory. If you want to use your current directory, just press the return key.

It will detect the languages used in your project and provide a percentage distribution of those languages in your project. You will now be presented with a list of bears that might be relevant to your project to choose from. Once you choose your bears you are done.

At the end, you should have a file named `.coafile` generated at the root of your project directory. This contains all the settings needed by coala to lint and fix your code. You can also open the .coafile in your favorite editor and edit the settings to suit your needs.

Once you have completed these steps just execute coala from your project's root:

```
$ coala
```

CHAPTER 6

Glob - Extended unix style pathname expansion

6.1 Using Glob Patterns

Suppose you want `SpaceConsistencyBear` to perform an analysis on a file `first.c`, you should use the command:

```
coala --files=src/first.c --bears=SpaceConsistencyBear --save
```

Note: If you don't know the functions of a bear or how to perform the analysis with a bear, you should go through Tutorial first.

Now, if you want all the `.c` files in a specific directory to be analysed, you can take help of glob patterns.

```
coala --files='src/*.c' --bears=SpaceConsistencyBear --save
```

Here, `*.c` matches all `.c` files in the `src` directory. Going further, if you want all `.c` as well as `.java` files to be analysed:

```
coala --files='src/*.(java|c)' --bears=SpaceConsistencyBear --save
```

If you want your `files` argument to match all directories and subdirectories, you can use `**` glob pattern for that.

```
files = **/*.c
```

Note: While using `files = **/*.c`, since we have used `/` in the glob pattern, all `.c` files at least one subdirectory below the root directory will be matched.

In coala, files and directories are specified by file name. To allow input of multiple files without requiring a large number of filenames, coala supports a number of wildcards. These are based on the unix-style glob syntax and they are *not* the same as regular expressions.

Note: Any glob that does not start with a `/` in Linux or a drive letter `X:` in Windows will be interpreted as a relative path. Please use comma separated values instead of absolute path globs that start with a glob expression.

6.2 Syntax

The special characters used in shell-style wildcards are:

PATTERN	MEANING
`[seq]`	Matches any character in seq. Cannot be empty. Any special character loses its special meaning in a set.
`[!seq]`	Matches any character not in seq. Cannot be empty. Any special character loses its special meaning in a set.
`(seq_a|seq_b)`	Matches either sequence_a or sequence_b as a whole. More than two or just one sequence can be given.
`?`	Matches any single character.
`*`	Matches everything but the directory separator.
`**`	Matches everything.

Note: If you're looking for a negation pattern to exclude paths, check out the `--ignore` argument or `ignore` .coafile option here.

6.3 Examples

6.3.1 `[seq]`

Matches any character in seq. Cannot be empty. Any special character loses its special meaning in a set.

Opening and closing brackets can be part of a set, although closing brackets have to be placed at the first position.

```
>>> from coalib.parsing.Globbing import fnmatch
>>> fnmatch("aaa", "a[abc]a")
True
>>> fnmatch("aaa", "a[bcd]a")
False
>>> fnmatch("aaa", "a[a]]a")
False
>>> fnmatch("aa]a", "a[a]]a")
True
>>> fnmatch("aaa", "a[]abc]a")
True
>>> fnmatch("aaa", "a[[a]a")
True
>>> fnmatch("a[a", "a[[a]a")
True
>>> fnmatch("a]a", "a[]]a")
True
>>> fnmatch("aa", "a[]a")
False
```

```
>>> fnmatch("a[]a", "a[]a")
True
```

6.3.2 [!seq]

Matches any character not in seq. Cannot be empty. Any special character loses its special meaning in a set.

```
>>> fnmatch("aaa", "a[!a]a")
False
>>> fnmatch("aaa", "a[!b]a")
True
>>> fnmatch("aaa", "a[b!b]a")
False
>>> fnmatch("a!a", "a[b!b]a")
True
>>> fnmatch("a!a", "a[!]a")
False
>>> fnmatch("a[!]a", "a[!]a")
True
```

6.3.3 (seq_a\|seq_b)

Matches either sequence_a or sequence_b as a whole. More than two or just one sequence can be given.

Parentheses cannot be part of an alternative, unless they are escaped by brackets. Parentheses that have no match are ignored as well as |-separators that are not inside matching parentheses.

```
>>> fnmatch("aXb", "a(X|Y)b")
True
>>> fnmatch("aYb", "a(X|Y)b")
True
>>> fnmatch("aZb", "a(X|Y)b")
False
>>> fnmatch("aXb", "(a(X|Y)b|c)")
True
>>> fnmatch("a", "a|b")
False
>>> fnmatch("a|b", "a|b")
True
>>> fnmatch("(aa", "(a(a|b)")
True
>>> fnmatch("a(a", "(a(a|b)")
False
>>> fnmatch("a(a", "(a[(]a|b)")
True
>>> fnmatch("aa", "a()a")
True
>>> fnmatch("", "(abc|)")
True
```

6.3.4 ?

Matches any single character.

```
>>> fnmatch("abc", "a?c")
True
>>> fnmatch("abbc", "a?c")
False
>>> fnmatch("a/c", "a?c")
True
>>> fnmatch("a\\c", "a?c")
True
>>> fnmatch("a?c", "a?c")
True
>>> fnmatch("ac", "a?c")
False
```

6.3.5 *

Matches everything but the directory separator.

Note: The directory separator is platform specific. `/` is never matched by `\*`. `\\` is matched on Linux, but not on Windows.

```
>>> fnmatch("abbc", "a*c")
True
>>> fnmatch("a/c", "a*c")
False
>>> fnmatch("ac", "a*c")
True
```

6.3.6 **

Matches everything.

```
>>> fnmatch("abbc", "a**c")
True
>>> fnmatch("a/c", "a**c")
True
```

CHAPTER 7

Exit Codes

The following is a list of coala's exit codes and their meanings:

- `0` - coala executed successfully but yielded no results.
- `1` - coala executed successfully but yielded results.
- `2` - Invalid arguments were passed to coala in the command line.
- `3` - The file collector exits with this code if an invalid pattern is passed to it.
- `4` - coala was executed with an unsupported version of python
- `5` - coala executed successfully. Results were found but patches to the results were applied successfully
- `13` - There is a conflict in the version of a dependency you have installed and the requirements of coala.
- `130` - A `KeyboardInterrupt` (`Ctrl+C`) was pressed during the execution of coala.
- `255` - Any other general errors.

CHAPTER 8

Git Hooks

This document is a guide on how to add coala as a git hook. Using a git hook coala can be executed automatically, ensuring your code follows your quality requirements.

8.1 Pre-Commit Hooks

The pre-commit hook can be used to run coala before every commit action. Hence, this does not allow any code not following the quality standards specified unless it's done by force.

To enable this, just create the file `.git/hooks/pre-commit` under your repository and add the lines:

```
#!/bin/sh
set -e
coala
```

You can also specify arguments like `-S autoapply=false` which tells coala to not apply any patch by itself. Or you can run specific sections with `coala <section_name>`.

See also:

Module ***Tutorial for Users*** Documentation on how to run coala which introduces the CLI arguments.

Module ***coafile Specification*** Documentation on how to configure coala using the coafile specification.

Note: If you allow coala to auto apply patches, it's recommended to add **.orig* to your .gitignore. coala creates these files while applying patches and they could be erroneously added to your commit.

This file needs to be executable. If it is not (or if you aren't sure), you can make it executable by running

```
$ chmod +x .git/hooks/pre-commit
```

and you're done! It will run every time before you commit, and prevent you from committing if the code has any errors.

CHAPTER 9

Shell Autocompletion

If you're using `bash` or `zsh` you can set them up to have tab completion for coala arguments and bear names.

Install `argcomplete`:

```
$ pip install argcomplete
```

After this you have to either activate it globally or modify your configuration.

If you're using `bash`, add the following to your `.bashrc`:

```
eval "$(register-python-argcomplete coala)"
```

If you're using `zsh`, add the following to your `.zshrc`:

```
autoload bashcompinit
bashcompinit
eval "$(register-python-argcomplete coala)"
```

CHAPTER 10

What is Docker?

> Docker is a tool designed to make it easier to create, deploy, and run applications by using containers. Containers allow a developer to package up an application with all of the parts it needs, such as libraries and other dependencies, and ship it all out as one package. By doing so, thanks to the container, the developer can rest assured that the application will run on any other Linux machine regardless of any customized settings that machine might have that could differ from the machine used for writing and testing the code.
>
> In a way, Docker is a bit like a virtual machine. But unlike a virtual machine, rather than creating a whole virtual operating system, Docker allows applications to use the same Linux kernel as the system that they're running on and only requires applications be shipped with things not already running on the host computer. This gives a significant performance boost and reduces the size of the application.

Cited from opensource.com.

10.1 What is a Docker Image and how is it different from a container?

An image is an immutable file which contains required binaries and libraries needed to make a container and a running instance of an image is called a container. Images are composed of layers of other images. Images are created when we run the `build` command of Docker and containers are formed from these images when we use the `run` command of Docker. There can be many containers for the same image.

For more information about Docker see the official documentation.

10.2 Installing Docker

Docker installation guide for various operating systems can be found in the official Docker installation instructions.

Note: Docker images are usually very large. Downloading or pushing them over low bandwidth connections can be very slow.

CHAPTER 11

coala as a Docker Image

We provide a `coala/base` docker image for your convenience, that has dependencies for most official bears already installed.

You can use the `coala/base` docker image to perform static code analysis on your code in the working directory, like this:

```
docker run -v=$(pwd):/app --workdir=/app coala/base coala --ci
```

See also:

See also https://hub.docker.com/r/coala/base/.

Note: The coala Docker image does not support Python 2 analysis.

You can add coala as alias for docker image, like this:

```
alias coala="docker run -ti -v $(pwd):/app --workdir=/app coala/base coala"
```

11.1 coala on GitLab CI

You can use the `coala/base` docker image to perform static code analysis on your code with a `.gitlab-ci.yml`, like this:

```
check_code:
  image: coala/base
  script:
  - pip install -r requirements.txt
  - coala --ci
```

Note: For more information about GitLab CI configuration, consult the official documentation.

11.2 Troubleshooting GitLab CI

You might experience DNS related difficulties with a private GitLab CI setup. The coala container might not be able to clone the repository if the GitLab server name is not resolvable.

When this is the case, the most straightforward workaround is to add a configuration line inside the `config.toml` configuration file for the `gitlab-ci-multi-runner` runner:

```
extra_hosts = ["my-gitlab.example.com:192.168.0.100"]
```

Please be aware that the most generic `dns` setting listed in the `gitlab-ci-multi-runner` documentation has been recently added and at the time of this writing is not available in official builds.

11.3 coala on Travis CI

You can use the `coala/base` docker image to perform static code analysis on your code with a `.travis.yml`, like this:

```
language: generic
services: docker
script: docker run -v=$(pwd):/app --workdir=/app coala/base coala --ci
```

Note: For more information about Travis CI configuration, consult the official documentation.

11.4 coala on Circle CI

You can use the `coala/base` docker image to perform static code analysis on your code with a `circle.yml`, like this:

```
machine:
  services:
    - docker

test:
  override:
    - docker run -v=$(pwd):/app --workdir=/app coala/base coala --ci
```

Note: For more information about Circle CI configuration, consult the official documentation.

CHAPTER 12

Coverage Installation Hints for macOS Users:

12.1 venv

Here we will be using `venv`, which is part of python's standard libary since python 3.3, to create a virtualenv for development.

12.1.1 1. Make sure you have installed Xcode and Homebrew.

12.1.2 2. Install Python 3.

For coala you will need to use Python 3, so you may simply use homebrew to install Python 3, or you could also *refer to the pyenv section* to install Python 3 while you can also maintain other python versions.

```
$ brew search python  # This should display Python 3
$ brew install python3
$ python3 --version   # To check the installed version
```

12.1.3 3. Create Virtual Environments with venv

```
# Create Virtual Env named myenv
$ python3 -m venv myenv

# This will create a folder named myenv in the
# current directory. To activate this environment just type
$ source myenv/bin/activate

# You can start Python 3 by typing:
$ python
```

12.1.4 4. Virtualenvwrapper with Python 3:

```
# Installation
$ pip3 install virtualenv
$ pip3 install virtualenvwrapper

# Folder to contain Virtual Environments
$ mkdir ~/.virtualenvs

# Add the following in ~/.bash_profile
$ export WORKON_HOME=~/.virtualenvs
$ source /usr/local/bin/virtualenvwrapper.sh

# Activate Changes
$ source ~/.bash_profile

# Get Python 3 path (python3_pth)
$ which python3

# Create a new virtual environment with Python 3
$ mkvirtualenv --python=python3_path myenv
```

12.1.5 Finally!

```
# Install python-coverage3 by
$ easy_install coverage
```

12.2 pyenv

Here we show how to use pyenv, which is simply a python installer that allows you to install multiple python versions and switch through them. After installation, you may also want to set up a virtual environment using `venv` or `virtualenv`.

12.2.1 1. Install pyenv with homebrew

```
$ brew update
$ brew install pyenv
```

12.2.2 2. Installing and using python with pyenv

To install python you simply have to do this (you don't have to install all of them) :

```
$ pyenv install 3.5.0
$ pyenv install 3.4.3
$ pyenv install 3.3.6
$ pyenv install 3.2.6
$ pyenv install 2.7.10
$ pyenv versions # lists the versions you have
```

To use a particular version you simply have to do this :

```
$ pyenv local 3.5.0
```

To know more about the available pyenv commands, please read their Command Reference.

Presently, while using pyenv, tests for version-file-read fail, if interested you can have a look here.

CHAPTER 13

Getting In Touch With Us

We are working hard to make coala a top choice for static code analysis. coala is a place where people from all over the world are part of a friendly, growing community. If you want to get in touch with us, you can do any of the following things:

- Join us on our gitter channel where we are very active and happy to help you!
- Subscribe to our mailing lists.
- Open an issue. Whether you want a new feature for coala or you have found a bug, just file an issue and we will check it out. In case of long discussion, you should create a new issue with clearly defined task!
- Give us feedback. If you think we're doing something useless or something amazing, let us know by dropping a message on gitter or a mail on our mailing lists!
- If you believe someone is violating the code of conduct , we ask that you report it by emailing `community AT coala DOT io`.

We appreciate any help. (Partially with words, partially with chocolate if you live near Hamburg or join us at conferences.)

Modularity, clean good code as well as a high usability for both users and developers of analyse routines (called bears) stand in the foreground of the development. We will not speed up our development if it needs sacrificing any aspect of quality.

CHAPTER 14

Frequently Asked Questions

This is a list of frequently asked questions, aiming to answer any possible questions by newcomers or even contributors.

Table of Contents

14.1 What does coala do (for me)?

coala is like a spell and grammar checker for source code. It offers one command and one configuration to lint all languages in your project.

In short coala does two things:

- Makes it easy to use existing static code analyzers such as clang, pylint(and many others) by unifying and simplifying the configs
- Makes it easy to write new routines by offering the interface part and everything but the actual analyzer routine. Write just the parameters your custom analysis needs and the analysis part, we'll take care of the rest

14.2 Can I Use coala in my Continuous Integration?

Yes! There's an argument *–ci* that runs the coala binary in non-interactive mode, which shows results cleanly in the log and returns error codes if something goes wrong. List of the exit codes and their meanings: Exit Codes.

14.3 Why did you choose the name?

coala stands for "COde AnaLysis Application", works well with animals and thus is well visualizable, it's easy to memorize.

14.4 Is there corporate backing behind coala? What are your intentions?

coala was founded for fun. coala was, is and will always be free software and is developed mostly by students and there's no corporate interest and no CLA. If you want to back us, contact us on our gitter channel.

14.5 What sort of analysis does coala do? What languages are supported?

A list of all analysis routines and supported languages is fully browsable.

For a top level view on what languages support what kind of analysis roughly, consult this link.

There are also generic bears, which can be applied language independently on your code. Their capabilities and information can be seen here.

14.6 What is a bear anyway?

A bear is a routine that is used by coala to check your code for lint issues. A group of these routines(bears) is used for defining certain quality requirements in various languages. A bear can analyse code for potential problems, calculate metrics and even provide corrections for your code. While many of these bears for various languages are shipped along with coala, you can easily write some of your own.

14.7 How do I get started with coala?

If you're looking to get started using coala, we have a *full tutorial* that will teach you everything you need to know to use coala.

If you're willing to contribute and become a part of our coalaian community, we have written a newcomers guide that will help you fix an issue on your own, just by following and understanding the indications. It is meant for newcomers, and it does not require you to have any precedent knowledge regarding coala.

14.8 How do I get in touch with the coala team?

We are very active on our gitter channel and will try to respond to any question in a matter of minutes. However, for a full list of how to get in touch with us, consult *this link*.

14.9 Installation is Failing! Help!

Don't panic!

Scroll down the error log, you will probably see something like *ValueError: ('Expected version spec in', 'appdirs ~=1.4.0', 'at', ' ~=1.4.0')* there. If not, ask us!

If so, you're probably on a Debian with an outdated pip that doesn't support our version specifiers yet. You have to create a virtual environment with a newer pip:

```
pip3 install virtualenv
virtualenv -p python3 ~/venvs/coala
. ~/venvs/coala/bin/activate
pip install -U pip
pip install coala-bears
```

should do the job. You have to activate this virtualenv on every terminal session you want to use coala though (tip: add it to bashrc!)

14.10 What are those things failing/passing on my Pull Request?

We use a few checks to make sure your Pull Request is ready to be merged into our master branch. Right now we use 7 of those checks:

- **review/gitmate/commit** Checks this particular commit has any new gitmate issues.
- **review/gitmate/pr** Checks whether your code respects our styling (PEP8), doesn't contain unneeded newlines, trailing whitespace, etc. Basically it is like running coala over your code, so to fix this, simply run `$ coala` before pushing! In case you have multiple commits, and the issue is in one of them, the status will still be failed, so be careful to run `$ coala` before making each commit.
- **review/gitmate/manual** This one is the only one that is manual, this can be given by any coala member and shows that the commit has been reviewed and has no problems, so it is ready to be merged. It can be done by commenting `ack commit_sha`. For more information about the whole process, we have it all documented here.
- **codecov/project** This one checks whether all your code is being tested. We cannot merge anything that may not work or may broke somewhere, so to avoid obvious bugs we use this. To fix it, write doctests or unittests for your functions / classes.
- **ci/circleCI** This is one of the two containers we use to continuously test the code. It basically runs all the tests and checks your code in a container, checking that the tests pass on the container. This one is for Linux, it runs Ubuntu 12.04.

- **continuous-integration/appveyor/pr** This one does the same as the one above, but for Windows, both 32 and 64bits versions.

14.11 What is coala Community Code of Conduct?

The coala team and community is made up of a mixture of professionals and volunteers from all over the world. Diversity is one of our huge strengths, but it can also lead to communication issues and unhappiness. To that end, we have a few ground rules that we ask people to adhere to. A guide to make it easier to enrich all of us and the technical communities in which we participate. This code of conduct applies to all spaces managed by the coala project.

If you believe someone is violating the code of conduct, we ask that you report it by emailing `community AT coala DOT io`. In general:

- **Be friendly and patient.**
- **Be welcoming.**
- **Be considerate.**
- **Be respectful.**
- **Be careful in the words that you choose.**
- **When we disagree, try to understand why.**

For more information refer Code of Conduct.

You might also want to look at our website.

CHAPTER 15

coala: Language Independent Code Analysis

coala provides a unified command-line interface for linting and fixing all your code, regardless of the programming languages you use.

With coala, users can create *rules and standards* to be followed in the source code. coala has a **user-friendly interface** that is completely customizable. It can be used in any environment and is completely modular.

coala has a set of official bears (plugins) for several languages, including popular languages such as **C/C++**, **Python**, **JavaScript**, **CSS**, **Java** and many more, in addition to some generic language independent algorithms. To learn more about the different languages supported and the bears themselves, click here.

Note: To see what coala can do for you and your language, take a look at our capabilities listing.

If you are here to use coala for your own projects, take a look at our *installation guide*.

If you want to start contributing to coala, you can follow our tutorial for newcomers which aims to get everyone to fix an issue by themselves.

Note: To contact us, always feel free to check our *Getting In Touch* page.

CHAPTER 16

What do I get?

16.1 As a User

coala allows you to simply check your code against certain quality requirements. The checking routines are named **Bears** in coala. You can easily define a simple project file to check your project with all bears either shipped with coala or ones you found in the internet and trust.

16.2 As a Developer

If you are not satisfied with the functionality given by the bears we provide, you can easily write your own bears. coala is written with easiness of extension in mind. That means: no big boilerplate, just write one small object with one routine, add the parameters you like and see how coala automates the organization of settings, user interaction and execution parallelization. You shouldn't need to care about anything else than just writing your algorithm!

See also:

Check out Writing Bears for more information on this.

To programmatically access coala's functionality, use the `--json` option. Use the `--format` option if you want to use a custom format string. Both of these arguments along with `--ci` argument run coala in non-interactive mode which is suitable for continuous integration.

16.3 Hint

`--json` makes the coala output easy to read and understand. It consists of a collection of name/value pairs usually represented by objects and an ordered list of values stored in arrays. You can read more about this format here.

`--format` has a linear, one line return. Its output can be easily parsed as it is fully customizable. This option will not show all the tested areas but those with issues. In case of no errors, `--format` will have no output.

CHAPTER 17

Status and Stability of the Project

We are currently working hard to make this project reality. coala is currently usable, in beta stage and already provides more features than most language dependent alternatives. Every single commit is fully reviewed and checked with various automated methods including our testsuite covering all branches. Our master branch is continuously prereleased to our users so you can rely on the upcoming release being rock stable.

If you want to see how the development progresses, check out coala and coala-bears.