# Contributing

## Local development

- The complete test suite depends on having at least the following installed

(possibly not a complete list)

- git (Version 2.24.0 or above is required to run pre-merge-commit tests)

- python2 (Required by a test which checks different python versions)

- python3 (Required by a test which checks different python versions)

- tox (or virtualenv)

- ruby + gem

- docker

- conda

- cargo (required by tests for rust dependencies)

- go (required by tests for go dependencies)

- swift

### Setting up an environment

This is useful for running specific tests. The easiest way to set this up

is to run:

1. `tox --devenv venv` (note: requires tox>=3.13)

2. `. venv/bin/activate` (or follow the [activation instructions] for your

platform)

This will create and put you into a virtualenv which has an editable

installation of pre-commit. Hack away! Running `pre-commit` will reflect

your changes immediately.

### Running a specific test

Running a specific test with the environment activated is as easy as:

`pytest tests -k test\_the\_name\_of\_your\_test`

### Running all the tests

Running all the tests can be done by running `tox -e py37` (or your

interpreter version of choice). These often take a long time and consume

significant cpu while running the slower node / ruby integration tests.

Alternatively, with the environment activated you can run all of the tests

using:

`pytest tests`

### Setting up the hooks

With the environment activated simply run `pre-commit install`.

## Documentation

Documentation is hosted at https://pre-commit.com

This website is controlled through

https://github.com/pre-commit/pre-commit.github.io

## Adding support for a new hook language

pre-commit already supports many [programming languages](https://pre-commit.com/#supported-languages)

to write hook executables with.

When adding support for a language, you must first decide what level of support

to implement. The current implemented languages are at varying levels:

- 0th class - pre-commit does not require any dependencies for these languages

as they're not actually languages (current examples: fail, pygrep)

- 1st class - pre-commit will bootstrap a full interpreter requiring nothing to

be installed globally (current examples: node, ruby)

- 2nd class - pre-commit requires the user to install the language globally but

will install tools in an isolated fashion (current examples: python, go, rust,

swift, docker).

- 3rd class - pre-commit requires the user to install both the tool and the

language globally (current examples: script, system)

"third class" is usually the easiest to implement first and is perfectly

acceptable.

Ideally the language works on the supported platforms for pre-commit (linux,

windows, macos) but it's ok to skip one or more platforms (for example, swift

doesn't run on windows).

When writing your new language, it's often useful to look at other examples in

the `pre\_commit/languages` directory.

It might also be useful to look at a recent pull request which added a

language, for example:

- [rust](https://github.com/pre-commit/pre-commit/pull/751)

- [fail](https://github.com/pre-commit/pre-commit/pull/812)

- [swift](https://github.com/pre-commit/pre-commit/pull/467)

### `language` api

here are the apis that should be implemented for a language

Note that these are also documented in [`pre\_commit/languages/all.py`](https://github.com/pre-commit/pre-commit/blob/master/pre\_commit/languages/all.py)

#### `ENVIRONMENT\_DIR`

a short string which will be used for the prefix of where packages will be

installed. For example, python uses `py\_env` and installs a `virtualenv` at

that location.

this will be `None` for 0th / 3rd class languages as they don't have an install

step.

#### `get\_default\_version`

This is used to retrieve the default `language\_version` for a language. If

one cannot be determined, return `'default'`.

You generally don't need to implement this on a first pass and can just use:

```python

get\_default\_version = helpers.basic\_default\_version

```

`python` is currently the only language which implements this api

#### `healthy`

This is used to check whether the installed environment is considered healthy.

This function should return `True` or `False`.

You generally don't need to implement this on a first pass and can just use:

```python

healthy = helpers.basic\_healthy

```

`python` is currently the only language which implements this api, for python

it is checking whether some common dlls are still available.

#### `install\_environment`

this is the trickiest one to implement and where all the smart parts happen.

this api should do the following things

- (0th / 3rd class): `install\_environment = helpers.no\_install`

- (1st class): install a language runtime into the hook's directory

- (2nd class): install the package at `.` into the `ENVIRONMENT\_DIR`

- (2nd class, optional): install packages listed in `additional\_dependencies`

into `ENVIRONMENT\_DIR` (not a required feature for a first pass)

#### `run\_hook`

This is usually the easiest to implement, most of them look the same as the

`node` hook implementation:

https://github.com/pre-commit/pre-commit/blob/160238220f022035c8ef869c9a8642f622c02118/pre\_commit/languages/node.py#L72-L74

[activation instructions]: https://virtualenv.pypa.io/en/latest/user\_guide.html#activators