# Contributing to UltiSnips

:+1::tada: First off, thanks for taking the time to contribute! :tada::+1:

This document will take you through the process of making your change, testing it, documenting it and sending it for review.

No new feature will be accepted without good test coverage, passing CI and proper documentation.

## Before you add a feature

UltiSnips is so rich on features that it borders on feature creep.

It is also an understaffed and undermaintained project.

Since every feature needs to be maintained forever, we are very careful about new ones.

Please create alignment before putting too much work into a novel idea.

There are several ways of doing this:

1. Open an issue to discuss your idea.

2. Open a PR with a hackish or minimal implementation, i.e. no tests and no docs.

3. Write a short (<= 1 page) design doc in a Gist or on Google Docs.

Should there be agreement that your feature idea adds enough value to offset the maintenance burden, you can go ahead and implement it, including tests and documentation.

## Testing

UltiSnips has a rigorous test suite and every new feature or bug fix is expected to come with a new test.

The overwhelming number of the > 500 test cases are integration tests.

Each test case sets up a full on-disk Vim configuration, including `.vimrc`, plugins and snippet definitions.

We then simulate a user typing out a test case by programmatically sending keys into a [tmux](https://github.com/tmux/tmux/wiki) terminal that runs Vim.

A test is a Python class in the `test` directory.

Some simple examples are in [test\_Expand.py](https://github.com/SirVer/ultisnips/blob/master/test/test\_Expand.py).

Each class contains at least

- a `keys` property that defines the key strokes taken,

- a `wanted` golden string that defines the expected output of the snippet, and

- a `snippets` list that defines the snippet that are in scope for the test case.

Each test types out a given set of key strokes and compares the resulting text in the Vim buffer to `wanted`.

### Running the test suite.

The basic process of running the suite is simple:

1. open a terminal and start a new tmux session in the current directory named

vim: `tmux new -s vim`. Do not type anything into the tmux session.

2. In a second terminal, run `./test\_all.py`.

To filter the tests that are executed, specify a pattern to be used to match the beginning of the test name.

For instance, the following will execute all tests that start with `SimpleExpand`:

$ ./test\_all.py SimpleExpand

Currently, the test suite only runs under Linux and Mac, not under Windows.

Contributions to make it work under Windows again would be very much appreciated.

#### Running using docker.

The problem with running tests on the system directly is that the user's environment can bleed into the test execution.

To avoid this problem, we strongly suggest running the tests inside of [Docker](https://www.docker.com/).

It is useful to think of Docker as a lightweight virtual machine, i.e. a way of running exactly the same OS and userland configuration on any machine.

UltiSnips comes with a [Makefile](https://github.com/SirVer/ultisnips/blob/master/Makefile) that makes the use of Docker easy.

First, build the image of the test environment (Vim 8.0, using Python 3):

$ make image\_repro

Now we can launch the image in a container and run the tmux session for testing.

$ make repro

... now inside container

# tmux new -s vim

The container will have the current directory mounted under `/src/UltiSnips`.

This means all changes you make to UltiSnips' sources will directly be represented inside the container and therefore available for testing.

In a second terminal we'll use `docker run` to get another shell in the already running container.

In this shell we can then trigger the test execution:

$ make shell\_in\_repro

... now inside container

# ./test\_all.py

## Documenting

User documentation goes into [`doc/UltiSnips.txt`](https://github.com/SirVer/ultisnips/blob/00\_contributing/doc/UltiSnips.txt).

Developer documentation should go into this file.

# Reproducing Bugs

Reproducing bugs is the hardest part of getting them fixed.

UltiSnips is usually used in complex interaction with other software and plugins.

This makes reproducing issues even harder.

Here is a process of creating a minimal viable reproduction case for your particular problem.

Having this available in a bug report will increase the chances of your issue being fixed tremendously.

1. Install [Docker](https://docs.docker.com/install/). It is useful to think of Docker as a lightweight virtual machine, i.e. a way of running exactly the same OS and userland configuration on any machine.

2. Build the image using `make image\_repro`.

3. Launch the image using `make repro`. This drops you into a shell where you can run `vim` to get to a vim instance configured for UltiSnips.

Now try to reproduce your issue.

Keep in mind that all your changes to the container are lost once you exit the shell.

You must edit the configuration outside the container and rebuild the image.

You can add snippets to `docker/snippets/`.

You can also copy more and more of your own `.vimrc` into `docker/docker\_vimrc.vim` until your issue reproduces.

Whenever you have edited `snippets` or `docker\_vimrc.vim` you need to rerun `make image\_repo && make repro`.

Once you have a minimal complete repro case ready,

1. fork UltiSnips,

2. commit your changes to the Vim configuration into a branch,

3. push the branch,

4. link it in the issue.