# Contributing

First read the overall project contributing guidelines. These are all

included in the qiskit documentation:

https://qiskit.org/documentation/contributing\_to\_qiskit.html

## Contributing to Qiskit Terra

In addition to the general guidelines there are specific details for

contributing to terra, these are documented below.

### Pull request checklist

When submitting a pull request and you feel it is ready for review,

please ensure that:

1. The code follows the code style of the project and successfully

passes the tests. For convenience, you can execute `tox` locally,

which will run these checks and report any issues.

2. The documentation has been updated accordingly. In particular, if a

function or class has been modified during the PR, please update the

\*docstring\* accordingly.

3. If it makes sense for your change that you have added new tests that

cover the changes.

4. Ensure that if your change has an end user facing impact (new feature,

deprecation, removal etc) that you have added a reno release note for that

change and that the PR is tagged for the changelog.

### Changelog generation

The changelog is automatically generated as part of the release process

automation. This works through a combination of the git log and the pull

request. When a release is tagged and pushed to github the release automation

bot looks at all commit messages from the git log for the release. It takes the

PR numbers from the git log (assuming a squash merge) and checks if that PR had

a `Changelog:` label on it. If there is a label it will add the git commit

message summary line from the git log for the release to the changelog.

If there are multiple `Changelog:` tags on a PR the git commit message summary

line from the git log will be used for each changelog category tagged.

The current categories for each label are as follows:

| PR Label | Changelog Category |

| -----------------------|--------------------|

| Changelog: Deprecation | Deprecated |

| Changelog: New Feature | Added |

| Changelog: API Change | Changed |

| Changelog: Removal | Removed |

| Changelog: Bugfix | Fixed |

### Release Notes

When making any end user facing changes in a contribution we have to make sure

we document that when we release a new version of qiskit-terra. The expectation

is that if your code contribution has user facing changes that you will write

the release documentation for these changes. This documentation must explain

what was changed, why it was changed, and how users can either use or adapt

to the change. The idea behind release documentation is that when a naive

user with limited internal knowledge of the project is upgrading from the

previous release to the new one, they should be able to read the release notes,

understand if they need to update their program which uses qiskit, and how they

would go about doing that. It ideally should explain why they need to make

this change too, to provide the necessary context.

To make sure we don't forget a release note or if the details of user facing

changes over a release cycle we require that all user facing changes include

documentation at the same time as the code. To accomplish this we use the

[reno](https://docs.openstack.org/reno/latest/) tool which enables a git based

workflow for writing and compiling release notes.

#### Adding a new release note

Making a new release note is quite straightforward. Ensure that you have reno

installed with::

pip install -U reno

Once you have reno installed you can make a new release note by running in

your local repository checkout's root::

reno new short-description-string

where short-description-string is a brief string (with no spaces) that describes

what's in the release note. This will become the prefix for the release note

file. Once that is run it will create a new yaml file in releasenotes/notes.

Then open that yaml file in a text editor and write the release note. The basic

structure of a release note is restructured text in yaml lists under category

keys. You add individual items under each category and they will be grouped

automatically by release when the release notes are compiled. A single file

can have as many entries in it as needed, but to avoid potential conflicts

you'll want to create a new file for each pull request that has user facing

changes. When you open the newly created file it will be a full template of

the different categories with a description of a category as a single entry

in each category. You'll want to delete all the sections you aren't using and

update the contents for those you are. For example, the end result should

look something like::

```yaml

features:

- |

Introduced a new feature foo, that adds support for doing something to

``QuantumCircuit`` objects. It can be used by using the foo function,

for example::

from qiskit import foo

from qiskit import QuantumCircuit

foo(QuantumCircuit())

- |

The ``qiskit.QuantumCircuit`` module has a new method ``foo()``. This is

the equivalent of calling the ``qiskit.foo()`` to do something to your

QuantumCircuit. This is the equivalent of running ``qiskit.foo()`` on

your circuit, but provides the convenience of running it natively on

an object. For example::

from qiskit import QuantumCircuit

circ = QuantumCircuit()

circ.foo()

deprecations:

- |

The ``qiskit.bar`` module has been deprecated and will be removed in a

future release. Its sole function, ``foobar()`` has been superseded by the

``qiskit.foo()`` function which provides similar functionality but with

more accurate results and better performance. You should update your calls

``qiskit.bar.foobar()`` calls to ``qiskit.foo()``.

```

You can also look at other release notes for other examples.

You can use any restructured text feature in them (code sections, tables,

enumerated lists, bulleted list, etc) to express what is being changed as

needed. In general you want the release notes to include as much detail as

needed so that users will understand what has changed, why it changed, and how

they'll have to update their code.

After you've finished writing your release notes you'll want to add the note

file to your commit with `git add` and commit them to your PR branch to make

sure they're included with the code in your PR.

##### Linking to issues

If you need to link to an issue or other github artifact as part of the release

note this should be done using an inline link with the text being the issue

number. For example you would write a release note with a link to issue 12345

as:

```yaml

fixes:

- |

Fixes a race condition in the function ``foo()``. Refer to

`#12345 <https://github.com/Qiskit/qiskit-terra/issues/12345>` for more

details.

```

#### Generating the release notes

After release notes have been added if you want to see what the full output of

the release notes. In general the output from reno that we'll get is a rst

(ReStructuredText) file that can be compiled by

[sphinx](https://www.sphinx-doc.org/en/master/). To generate the rst file you

use the ``reno report`` command. If you want to generate the full terra release

notes for all releases (since we started using reno during 0.9) you just run::

reno report

but you can also use the ``--version`` argument to view a single release (after

it has been tagged::

reno report --version 0.9.0

At release time ``reno report`` is used to generate the release notes for the

release and the output will be submitted as a pull request to the documentation

repository's [release notes file](

https://github.com/Qiskit/qiskit/blob/master/docs/release\_notes.rst)

#### Building release notes locally

Building The release notes are part of the standard qiskit-terra documentation

builds. To check what the rendered html output of the release notes will look

like for the current state of the repo you can run: `tox -edocs` which will

build all the documentation into `docs/\_build/html` and the release notes in

particular will be located at `docs/\_build/html/release\_notes.html`

## Installing Qiskit Terra from source

Please see the [Installing Qiskit Terra from

Source](https://qiskit.org/documentation/contributing\_to\_qiskit.html#installing-terra-from-source)

section of the Qiskit documentation.

### Test

Once you've made a code change, it is important to verify that your change

does not break any existing tests and that any new tests that you've added

also run successfully. Before you open a new pull request for your change,

you'll want to run the test suite locally.

The easiest way to run the test suite is to use

[\*\*tox\*\*](https://tox.readthedocs.io/en/latest/#). You can install tox

with pip: `pip install -U tox`. Tox provides several advantages, but the

biggest one is that it builds an isolated virtualenv for running tests. This

means it does not pollute your system python when running. Additionally, the

environment that tox sets up matches the CI environment more closely and it

runs the tests in parallel (resulting in much faster execution). To run tests

on all installed supported python versions and lint/style checks you can simply

run `tox`. Or if you just want to run the tests once run for a specific python

version: `tox -epy37` (or replace py37 with the python version you want to use,

py35 or py36).

If you just want to run a subset of tests you can pass a selection regex to

the test runner. For example, if you want to run all tests that have "dag" in

the test id you can run: `tox -epy37 -- dag`. You can pass arguments directly to

the test runner after the bare `--`. To see all the options on test selection

you can refer to the stestr manual:

https://stestr.readthedocs.io/en/stable/MANUAL.html#test-selection

If you want to run a single test module, test class, or individual test method

you can do this faster with the `-n`/`--no-discover` option. For example:

to run a module:

```

tox -epy37 -- -n test.python.test\_examples

```

or to run the same module by path:

```

tox -epy37 -- -n test/python/test\_examples.py

```

to run a class:

```

tox -epy37 -- -n test.python.test\_examples.TestPythonExamples

```

to run a method:

```

tox -epy37 -- -n test.python.test\_examples.TestPythonExamples.test\_all\_examples

```

Alternatively there is a makefile provided to run tests, however this

does not perform any environment setup. It also doesn't run tests in

parallel and doesn't provide an option to easily modify the tests run.

For executing the tests with the makefile, a `make test` target is available.

The execution of the tests (both via the make target and during manual

invocation) takes into account the `LOG\_LEVEL` environment variable. If

present, a `.log` file will be created on the test directory with the

output of the log calls, which will also be printed to stdout. You can

adjust the verbosity via the content of that variable, for example:

Linux and Mac:

``` {.bash}

$ cd out

out$ LOG\_LEVEL="DEBUG" ARGS="-V" make test

```

Windows:

``` {.bash}

$ cd out

C:\..\out> set LOG\_LEVEL="DEBUG"

C:\..\out> set ARGS="-V"

C:\..\out> make test

```

For executing a simple python test manually, we don\'t need to change

the directory to `out`, just run this command:

Linux and Mac:

``` {.bash}

$ LOG\_LEVEL=INFO python -m unittest test/python/circuit/test\_circuit\_operations.py

```

Windows:

``` {.bash}

C:\..\> set LOG\_LEVEL="INFO"

C:\..\> python -m unittest test/python/circuit/test\_circuit\_operations.py

```

##### Test Skip Options

How and which tests are executed is controlled by an environment

variable, `QISKIT\_TESTS`:

Option | Description | Default

------ | ----------- | -------

`run\_slow` | It runs tests tagged as \*slow\*. | `False`

It is possible to provide more than one option separated with commas.

Alternatively, the `make test\_ci` target can be used instead of

`make test` in order to run in a setup that replicates the configuration

we used in our CI systems more closely.

### Development Cycle

The development cycle for qiskit-terra is all handled in the open using

the project boards in Github for project management. We use milestones

in Github to track work for specific releases. The features or other changes

that we want to include in a release will be tagged and discussed in Github.

As we're preparing a new release we'll document what has changed since the

previous version in the release notes.

### Branches

\* `master`:

The master branch is used for development of the next version of qiskit-terra.

It will be updated frequently and should not be considered stable. The API

can and will change on master as we introduce and refine new features.

\* `stable/\*` branches:

Branches under `stable/\*` are used to maintain released versions of qiskit-terra.

It contains the version of the code corresponding to the latest release for

that minor version on pypi. For example, stable/0.8 contains the code for the

0.8.2 release on pypi. The API on these branches are stable and the only changes

merged to it are bugfixes.

### Release cycle

When it is time to release a new minor version of qiskit-terra we will:

1. Create a new tag with the version number and push it to github

2. Change the `master` version to the next release version.

The release automation processes will be triggered by the new tag and perform

the following steps:

1. Create a stable branch for the new minor version from the release tag

on the `master` branch

2. Build and upload binary wheels to pypi

3. Create a github release page with a generated changelog

4. Generate a PR on the meta-repository to bump the terra version and

meta-package version.

The `stable/\*` branches should only receive changes in the form of bug

fixes.