Testing

Mojo includes a framework for developing and executing unit tests. The framework also supports testing code examples in the documentation strings (also known as *docstrings*) of your API references. The Mojo testing framework consists of a set of assertions defined as part of the Mojo standard library and the `mojo test` command line tool.

Get started

Let's start with a simple example of writing and running Mojo tests.

1. Write tests

For your first example of using the Mojo testing framework, create a file named `test_quickstart.mojo` containing the following code:

```
# Content of test_quickstart.mojo
from testing import assert_equal

def inc(n: Int) -> Int:
    return n + 1

def test_inc_zero():
    # This test contains an intentional logical error to show an example of
    # what a test failure looks like at runtime.
    assert_equal(inc(0), 0)

def test_inc_one():
    assert_equal(inc(1), 2)
```

In this file, the `inc()` function is the test *target*. The functions whose names begin with `test_` are the tests. Usually the target should be in a separate source file from its tests, but you can define them in the same file for this simple example.

A test function *fails* if it raises an error when executed, otherwise it *passes*. The two tests in this example use the `assert_equal()` function, which raises an error if the two values provided are not equal.

> **Note**
>
> The implementation of `test_inc_zero()` contains an intentional logical error so that you can see an example of a failed test when you execute it in the next step of this tutorial.

2. Execute tests

Then in the directory containing the file, execute the following command in your shell:

```
$ mojo test test_quickstart.mojo
```

You should see output similar to this (note that this example elides the full filesystem paths from the output shown):

```
Testing Time: 1.193s

Total Discovered Tests: 2

Passed : 1 (50.00%)
Failed : 1 (50.00%)
Skipped: 0 (0.00%)

******************** Failure: 'ROOT_DIR/test_quickstart.mojo::test_inc_zero()'
********************

Unhandled exception caught during execution

Error: At ROOT_DIR/test_quickstart.mojo:8:17: AssertionError: `left == right` comparison
failed:
   left: 1
  right: 0

********************
```

The output starts with a summary of the number of tests discovered, passed, failed, and skipped. Following that, each failed test is reported along with its error message.

Next steps

- The `testing` module describes the assertion functions available to help implement tests.
- Writing unit tests shows how to write unit tests and organize them into test files.
- The `mojo test` command describes how to execute and collect lists of tests.

- Writing API documentation tests discusses how to use the Mojo testing framework to test code examples in your API documentation.

The `testing` module

The Mojo standard library includes a `testing` module that defines several assertion functions for implementing tests. Each assertion returns `None` if its condition is met or raises an error if it isn't.

- `assert_true()`: Asserts that the input value is `True`.
- `assert_false()`: Asserts that the input value is `False`.
- `assert_equal()`: Asserts that the input values are equal.
- `assert_not_equal()`: Asserts that the input values are not equal.
- `assert_almost_equal()`: Asserts that the input values are equal up to a tolerance.

The boolean assertions report a basic error message when they fail.

```
from testing import *
assert_true(False)
```

```
Unhandled exception caught during execution

Error: At Expression [1] wrapper:14:16: AssertionError: condition was unexpectedly False
```

Each function also accepts an optional `msg` keyword argument for providing a custom message to include if the assertion fails.

```
assert_true(False, msg="paradoxes are not allowed")
```

```
Unhandled exception caught during execution

Error: At Expression [2] wrapper:14:16: AssertionError: paradoxes are not allowed
```

For comparing floating point values you should use `assert_almost_equal()`, which allows you to specify either an absolute or relative tolerance.

```
result = 10 / 3
assert_almost_equal(result, 3.33, atol=0.001, msg="close but no cigar")
```

```
Unhandled exception caught during execution
```

```
Error: At Expression [3] wrapper:15:24: AssertionError: 3.3333333333333335 is not close to 3.330000
```

The testing module also defines a context manager, `assert_raises()`, to assert that a given code block correctly raises an expected error.

```
def inc(n: Int) -> Int:
    if n == Int.MAX:
         raise Error("inc overflow")
    return n + 1

print("Test passes because the error is raised")
with assert_raises():
    _ = inc(Int.MAX)

print("Test fails because the error isn't raised")
with assert_raises():
    _ = inc(Int.MIN)
```

```
Unhandled exception caught during execution

Test passes because the error is raised
Test fails because the error isn't raised
Error: AssertionError: Didn't raise at Expression [4] wrapper:18:23
```

> **Note**
>
> The example above assigns the return value from `inc()` to a *discard pattern*. Without it, the Mojo compiler detects that the return value is unused and optimizes the code to eliminate the function call.

You can also provide an optional `contains` argument to `assert_raises()` to indicate that the test passes only if the error message contains the substring specified. Other errors are propagated, failing the test.

```
print("Test passes because the error contains the substring")
with assert_raises(contains="required"):
    raise Error("missing required argument")

print("Test fails because the error doesn't contain the substring")
with assert_raises(contains="required"):
    raise Error("invalid value")
```

```
Unhandled exception caught during execution

Test passes because the error contains the substring
```

```
Test fails because the error doesn't contain the substring
Error: invalid value
```

Writing unit tests

A Mojo unit test is simply a function that fulfills all of these requirements:

- Has a name that starts with `test_`.
- Accepts no arguments.
- Returns either `None` or a value of type `object`.
- Raises an error to indicate test failure.
- Is defined at the module scope, not as a Mojo struct method.

You can use either `def` or `fn` to define a test function. Because a test function always raises an error to indicate failure, any test function defined using `fn` must include the `raises` declaration.

Generally, you should use the assertion utilities from the Mojo standard library `testing` module to implement your tests. You can include multiple related assertions in the same test function. However, if an assertion raises an error during execution then the test function returns immediately, skipping any subsequent assertions.

You must define your Mojo unit tests in Mojo source files named with a `test` prefix or suffix. You can organize your test files within a directory hierarchy, but the test files must not be part of a Mojo package (that is, the test directories should not contain `__init__.mojo` files).

Here is an example of a test file containing three tests for functions defined in a source module named `my_target_module` (which is not shown here).

```
# File: test_my_target_module.mojo

from my_target_module import convert_input, validate_input
from testing import assert_equal, assert_false, assert_raises, assert_true

def test_validate_input():
        assert_true(validate_input("good"), msg="'good' should be valid input")
        assert_false(validate_input("bad"), msg="'bad' should be invalid input")

def test_convert_input():
        assert_equal(convert_input("input1"), "output1")
        assert_equal(convert_input("input2"), "output2")

def test_convert_input_error():
        with assert_raises():
                _ = convert_input("garbage")
```

The unique identity of a unit test consists of the path of the test file and the name of the test function, separated by `::`. So the test IDs from the example above are:

- `test_my_target_module.mojo::test_validate_input()`
- `test_my_target_module.mojo::test_convert_input()`
- `test_my_target_module.mojo::test_convert_error()`

The `mojo test` command

The `mojo` command line interface includes the `mojo test` command for running tests or collecting a list of tests.

Running tests

By default, the `mojo test` command runs the tests that you specify using one of the following:

- A single test ID with either an absolute or relative file path, to run only that test.
- A single absolute or relative file path, to run all tests in that file.
- A single absolute or relative directory path, to recurse through that directory hierarchy and run all tests found.

If needed, you can optionally use the `-I` option one or more times to append additional paths to the list of directories searched to import Mojo modules and packages. For example, consider a project with the following directory structure:

```
.
├── src
│   ├── example.mojo
│   └── my_math
│       ├── __init__.mojo
│       └── utils.mojo
└── test
    └── my_math
        ├── test_dec.mojo
        └── test_inc.mojo
```

From the project root directory, you could execute all of the tests in the `test` directory like this:

```
$ mojo test -I src test
Testing Time: 3.433s

Total Discovered Tests: 4

Passed : 4 (100.00%)
```

```
Failed : 0 (0.00%)
Skipped: 0 (0.00%)
```

You could run the tests contained in only the `test_dec.mojo` file like this:

```
$ mojo test -I src test/my_math/test_dec.mojo
Testing Time: 1.175s

Total Discovered Tests: 2

Passed : 2 (100.00%)
Failed : 0 (0.00%)
Skipped: 0 (0.00%)
```

And you could run a single test from a file by providing its fully qualified ID like this:

```
$ mojo test -I src 'test/my_math/test_dec.mojo::test_dec_valid()'
Testing Time: 0.66s

Total Discovered Tests: 1

Passed : 1 (100.00%)
Failed : 0 (0.00%)
Skipped: 0 (0.00%)
```

Collecting a list of tests

By including the `--collect-only` or `--co` option, you can use `mojo test` to discover and print a list of tests.

As an example, consider the project structure shown in the Running tests section. The following command produces a list of all of the tests defined in the `test` directory hierarchy.

```
$ mojo test --co test
```

The output shows the hierarchy of directories, test files, and individual tests (note that this example elides the full filesystem paths from the output shown):

```
<ROOT_DIR/test/my_math>
  <ROOT_DIR/test/my_math/test_dec.mojo>
    <ROOT_DIR/test/my_math/test_dec.mojo::test_dec_valid()>
    <ROOT_DIR/test/my_math/test_dec.mojo::test_dec_min()>
  <ROOT_DIR/test/my_math/test_inc.mojo>
```

```
    <ROOT_DIR/test/my_math/test_inc.mojo::test_inc_valid()>
    <ROOT_DIR/test/my_math/test_inc.mojo::test_inc_max()>
```

Producing JSON formatted output

By default `mojo test` produces concise, human-readable output. Alternatively you can produce JSON formatted output more suitable for input to other tools by including the `--diagnostic-format json` option.

For example, you could run the tests in the `test_quickstart.mojo` file shown in the Get started section with JSON formatted output using this command:

```
$ mojo test --diagnostic-format json test_quickstart.mojo
```

The output shows the detailed results for each individual test and summary results (note that this example elides the full filesystem paths from the output shown):

```
{
  "children": [
    {
      "duration_ms": 60,
      "error": "Unhandled exception caught during execution",
      "kind": "executionError",
      "stdErr": "",
      "stdOut": "Error: At ROOT_DIR/test_quickstart.mojo:8:17: AssertionError: `left ==
right` comparison failed:\r\n   left: 1\r\n  right: 0\r\n",
      "testID": "ROOT_DIR/test_quickstart.mojo::test_inc_zero()"
    },
    {
      "duration_ms": 51,
      "error": "",
      "kind": "success",
      "stdErr": "",
      "stdOut": "",
      "testID": "ROOT_DIR/test_quickstart.mojo::test_inc_one()"
    }
  ],
  "duration_ms": 1171,
  "error": "",
  "kind": "executionError",
  "stdErr": "",
  "stdOut": "",
  "testID": "ROOT_DIR/test_quickstart.mojo"
}
```

You can also produce JSON output for test collection as well. As an example, consider the project structure shown in the Running tests section. The following command collects a list in JSON format of all of the tests defined in the `test` directory hierarchy:

```
$ mojo test --diagnostic-format json --co test
```

The output would appear as follows (note that this example elides the full filesystem paths from the output shown):


```
{
  "children": [
    {
      "children": [
        {
          "id": "ROOT_DIR/test/my_math/test_dec.mojo::test_dec_valid()",
          "location": {
            "endColumn": 5,
            "endLine": 5,
            "startColumn": 5,
            "startLine": 5
          }
        },
        {
          "id": "ROOT_DIR/test/my_math/test_dec.mojo::test_dec_min()",
          "location": {
            "endColumn": 5,
            "endLine": 9,
            "startColumn": 5,
            "startLine": 9
          }
        }
      ],
      "id": "ROOT_DIR/test/my_math/test_dec.mojo"
    },
    {
      "children": [
        {
          "id": "ROOT_DIR/test/my_math/test_inc.mojo::test_inc_valid()",
          "location": {
            "endColumn": 5,
            "endLine": 5,
            "startColumn": 5,
            "startLine": 5
          }
        },
        {
          "id": "ROOT_DIR/test/my_math/test_inc.mojo::test_inc_max()",
          "location": {
            "endColumn": 5,
            "endLine": 9,
```

```
            "startColumn": 5,
            "startLine": 9
          }
        }
      ],
      "id": "ROOT_DIR/test/my_math/test_inc.mojo"
    }
  ],
  "id": "ROOT_DIR/test/my_math"
}
```

Writing API documentation tests

The Mojo testing framework also supports testing code examples that you include in docstrings. This helps to ensure that the code examples in your API documentation are correct and up to date.

Identifying executable code

The Mojo testing framework requires you to explicitly identify the code blocks that you want it to execute.

In a Mojo docstring, a fenced code block delimited by standard triple-backquotes is a *display-only* code block. It appears in the API documentation, but `mojo test` does not identify it as a test or attempt to execute any of the code in the block.

````
""" Non-executable code block example.

The generated API documentation includes all lines of the following code block,
but `mojo test` does not execute any of the code in it.

```
# mojo test does NOT execute any of this code block
a = 1
print(a)
```
"""
````

In contrast, a fenced code block that starts with the line ```` ```mojo ```` not only appears in the API documentation, but `mojo test` treats it as an executable test. The test fails if the code raises any error, otherwise it passes.

````
""" Executable code block example.

The generated API documentation includes all lines of the following code block
*and* `mojo test` executes it as a test.
````

```mojo
from testing import assert_equals

b = 2
assert_equals(b, 2)
```
"""
```

Sometimes you might want to execute a line of code as part of the test but *not* display that line in the API documentation. To achieve this, prefix the line of code with `%#`. For example, you could use this technique to omit `import` statements and assertion functions from the documentation.

```
""" Executable code block example with some code lines omitted from output.

The generated API documentation includes only the lines of code that do *not*
start with `%#`. However, `mojo test` executes *all* lines of code.

```mojo
%# from testing import assert_equal
c = 3
print(c)
%# assert_equal(c, 3)
```
"""
```

## Documentation test suites and scoping

The Mojo testing framework treats each docstring as a separate *test suite*. In other words, a single test suite could correspond to the docstring for an individual package, module, function, struct, struct method, etc.

Each executable code block within a given docstring is a single test of the same test suite. The `mojo test` command executes the tests of a test suite sequentially in the order that they appear within the docstring. If a test within a particular test suite fails, then all subsequent tests within the same test suite are skipped.

All tests within the test suite execute in the same scope, and test execution within that scope is stateful. This means, for example, that a variable created within one test is then accessible to subsequent tests in the same test suite.

```
""" Stateful example.

Assign 1 to the variable `a`:

```mojo
%# from testing import assert_equal
a = 1
%# assert_equal(a, 1)
```

```
Then increment the value of `a` by 1:

```mojo
a += 1
%# assert_equal(a, 2)
```
"""
```

> **Note**
>
> Test suite scopes do *not* nest. In other words, the test suite scope of a module is completely independent of the test suite scope of a function or struct defined within that module. For example, this means that if a module's test suite creates a variable, that variable is *not* accessible to a function's test suite within the same module.

Documentation test identifiers

The format of a documentation test identifier is `<path>@<test-suite>::<test>`. This is best explained by an example. Consider the project structure shown in the Running tests section. The source files in the `src` directory might contain docstrings for the `my_math` package, the `utils.mojo` module, and the individual functions within that module. You could collect the full list of tests by executing:

```
mojo test --co src
```

The output shows the hierarchy of directories, test files, and individual tests (note that this example elides the full filesystem paths from the output shown):

```
<ROOT_DIR/src/my_math>
  <ROOT_DIR/src/my_math/__init__.mojo>
    <ROOT_DIR/src/my_math/__init__.mojo@__doc__>
      <ROOT_DIR/src/my_math/__init__.mojo@__doc__::0>
      <ROOT_DIR/src/my_math/__init__.mojo@__doc__::1>
  <ROOT_DIR/src/my_math/utils.mojo>
    <ROOT_DIR/src/my_math/utils.mojo@__doc__>
      <ROOT_DIR/src/my_math/utils.mojo@__doc__::0>
    <ROOT_DIR/src/my_math/utils.mojo@inc(stdlib\3A\3Abuiltin\3A\3Aint\3A\3AInt).__doc__>
      <ROOT_DIR/src/my_math/utils.mojo@inc(stdlib\3A\3Abuiltin\3A\3Aint\3A\3AInt).__doc__::0>
      <ROOT_DIR/src/my_math/utils.mojo@inc(stdlib\3A\3Abuiltin\3A\3Aint\3A\3AInt).__doc__::1>
    <ROOT_DIR/src/my_math/utils.mojo@dec(stdlib\3A\3Abuiltin\3A\3Aint\3A\3AInt).__doc__>
      <ROOT_DIR/src/my_math/utils.mojo@dec(stdlib\3A\3Abuiltin\3A\3Aint\3A\3AInt).__doc__::0>
      <ROOT_DIR/src/my_math/utils.mojo@dec(stdlib\3A\3Abuiltin\3A\3Aint\3A\3AInt).__doc__::1>
```

Several different test suites appear in this result:

Test suite scope	File	Test suite name
Package	`src/my_math/__init__.mojo`	`__doc__`
Module	`src/my_math/utils.mojo`	`__doc__`
Function	`src/my_math/utils.mojo`	`inc(stdlib\3A\3Abuiltin\3A\3Aint\3A\3AInt).__doc__`

Then within a specific test suite, tests are numbered sequentially in the order they appear in the docstring, starting with 0.

Was this page helpful?

Edit this page