Ward

Release 0.56.0b0

Darren Burns

May 24, 2021

USER GUIDE

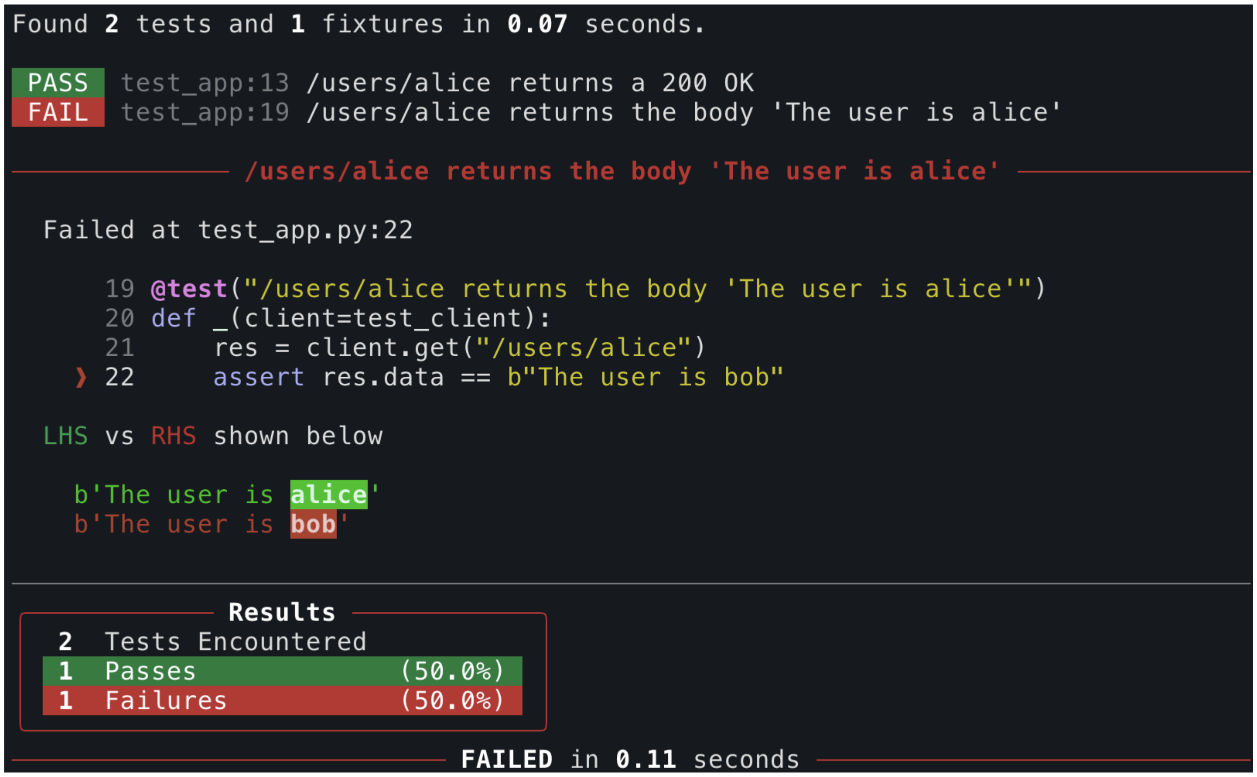
Ward is a modern test framework for Python with a focus on productivity and readability.

CHAPTER

ONE

FEATURES

- Describe your tests using strings instead of function names
- Use plain `assert` statements, with no need to remember `assert*` method names
- Beautiful output that focuses on readability
- Supported on MacOS, Linux, and Windows
- Manage test dependencies using a simple but powerful *fixture system*
- Support for *testing async code*
- *Parameterised testing* allows you to run a single test on multiple inputs
- Configurable with *pyproject.toml*, but works out-of-the-box with sensible defaults
- Speedy – Ward's suite of ~300 tests completes in ~0.4 seconds on my machine



```
Found 2 tests and 1 fixtures in 0.07 seconds.

 PASS   test_app:13 /users/alice returns a 200 OK
 FAIL   test_app:19 /users/alice returns the body 'The user is alice'

────────────── /users/alice returns the body 'The user is alice' ──────────────

  Failed at test_app.py:22

      19 @test("/users/alice returns the body 'The user is alice'")
      20 def _(client=test_client):
      21     res = client.get("/users/alice")
    ❯ 22     assert res.data == b"The user is bob"

  LHS vs RHS shown below

    b'The user is alice'
    b'The user is bob'

────────────────────────────────────────────────────────────────────────────────
 ┌──────────── Results ────────────┐
 │ 2  Tests Encountered            │
 │ 1  Passes              (50.0%)  │
 │ 1  Failures            (50.0%)  │
 └─────────────────────────────────┘
──────────────────────────── FAILED in 0.11 seconds ────────────────────────────
```

CHAPTER

TWO

INSTALLATION

Ward is available on PyPI, and can be installed with `pip install ward` (Python 3.6+ required).

CHAPTER

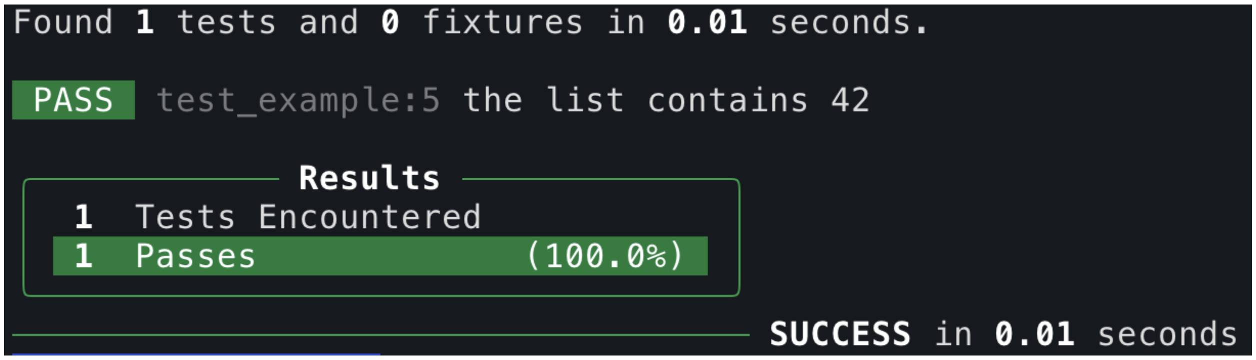
THREE

A QUICK TASTE

Here's a simple example of a test written using Ward:

```
# file: test_example.py
from ward import test

@test("the list contains 42")
def _():
    assert 42 in [-21, 42, 999]
```

To run the test, simply run `ward` in your terminal, and Ward will let you know how it went:



```
Found 1 tests and 0 fixtures in 0.01 seconds.

 PASS  test_example:5 the list contains 42

  ─────────── Results ───────────
     1  Tests Encountered
     1  Passes              (100.0%)

───────────────────── SUCCESS in 0.01 seconds
```

3.1 Writing Tests

3.1.1 Descriptive testing

Tests aren't only a great way of ensuring your code behaves correctly, they're also a fantastic form of documentation. Therefore, a test framework should make describing your tests in a clear and concise manner as simple as possible.

Ward lets you describe your tests using strings, allowing you to be as descriptive as you'd like:

```
from ward import test

@test("simple addition")
def _():
    assert 1 + 2 == 3
```

The description of a test is a format string, and may refer to any of the parameters (variables or fixtures) present in the test signature. This makes it easy to keep your test data and test descriptions in sync:

```python
@fixture
def three():
    yield 3

@test("{a} + {b} == {result}")
def _(a=1, b=2, result=three):
    assert a + b == result
```

During the test run, Ward will print the test description to the console.

Tests will only be collected from modules with names that start with *"test_"* or end with *"_test"*.

3.1.2 Tagging tests

You can tag tests using the `tags` keyword argument of the `@test` decorator:

```python
@test("simple addition", tags=["unit", "regression"])
def _():
    assert 1 + 2 == 3
```

Tags provide a powerful means of grouping tests and associating queryable metadata with them.

When running your tests, you can filter which ones you want to run using tag expressions.

Here are some ways you could use tags:

- Linking a test to a ticket from an issue tracker: *"BUG-123"*, *"PULL-456"*, etc.
- Describe what type of test it is: *"small"*, *"medium"*, *"big"*, *"unit"*, *"integration"*, etc.
- Specify which endpoint your test calls: *"/users"*, *"/tweets"*, etc.
- Specify which platform a test targets: *"windows"*, *"unix"*, *"ios"*, *"android"*

With your tests tagged you can now run only the tests you care about. To ask Ward to run only integration tests which target any mobile platform, you might invoke it like so:

`ward --tags "integration and (ios or android)"`

For a deeper look into tag expressions, see the [running tests](/guide/running-tests) page.

3.1.3 Using `assert` statements

Ward lets you use plain `assert` statements when writing your tests, but gives you considerably more information should the assertion fail than a typical *assert* statement. It does this by modifying the abstract syntax tree (AST) of any collected tests. Occurrences of the *assert* statement are replaced with a function call, depending on which comparison operator was used.

Currently, Ward only rewrites `assert` statements that appear directly in the body of your tests. If you use helper methods that contain `assert` statements and would like detailed output, you can use the helper `assert_{op}` methods from `ward.expect`.

3.1.4 Parameterised testing

A parameterised test is where you define a single test that runs multiple times, with different arguments being injected on each run.

The simplest way to parameterise tests in Ward is to write your test inside a loop. In each iteration of the loop, you can pass different values into the test:

```
for lhs, rhs, res in [
    (1, 1, 2),
    (2, 3, 5),
]:
    @test("simple addition")
    def _(left=lhs, right=rhs, result=res):
        assert left + right == result
```

You can also make a reference to a fixture and Ward will resolve and inject it:

```
@fixture
def five():
    yield 5

for lhs, rhs, res in [
    (1, 1, 2),
    (2, 3, five),
]:
    @test("simple addition")
    def _(left=lhs, right=rhs, result=res):
        assert left + right == result
```

Ward also supports parameterised testing by allowing multiple fixtures or values to be bound as a keyword argument using the `each function:

```
from ward import each, fixture, test

@fixture
def six():
    return 6

@test("an example of parameterisation")
def _(
    a=each(1, 2, 3),
    b=each(2, 4, six),
):
    assert a * 2 == b
```

Although the example above is written as a single test, Ward will generate and run 3 distinct tests from it at run-time: one for each item passed into *each*.

The variables `a` and `b` take the values `a=1` and `b=2` in the first test, `a=2` and `b=4` in the second test, and the third test will be passed the values `a=3` and `b=6`.

If any of the items inside `each` is a fixture, that fixture will be resolved and injected. Each of the test runs are considered *unique tests* from a fixture scoping perspective.

Warning: All occurrences of each in a test signature must contain the same number of arguments.

Using each in a test signature doesn't stop you from injecting other fixtures as normal.:

```python
from ward import each, fixture, test

@fixture
def book_api():
   return BookApi()

@test("BookApi.get_book returns the correct book given an ISBN")
def _(
   api=book_api,
   isbn=each("0765326353", "0765326361", "076532637X"),
   name=each("The Way of Kings", "Words of Radiance", "Oathbringer"),
)
   book: Book = api.get_book(isbn)
   assert book.name == name
```

Ward will expand the parameterised test above into 3 distinct tests.

In other words, the single parameterised test above is functionally equivalent to the 3 tests shown below.:

```python
@test("[1/3] BookApi.get_book returns the correct book given an ISBN")
def _(
   api=book_api,
   isbn="0765326353",
   name="The Way of Kings",
)
   book: Book = api.get_book(isbn)
   assert book.name == name

@test("[2/3] BookApi.get_book returns the correct book given an ISBN")
def _(
   api=book_api,
   isbn="0765326361",
   name="Words of Radiance",
)
   book: Book = api.get_book(isbn)
   assert book.name == name

@test("[3/3] BookApi.get_book returns the correct book given an ISBN")
def _(
   api=book_api,
   isbn="076532637X",
   name="Oathbringer",
)
   book: Book = api.get_book(isbn)
   assert book.name == name
```

If you'd like to use the same book_api instance across each of the three generated tests, you'd have to increase its scope to module or global.

Currently, each can only be used in the signature of *tests*.

3.1.5 Checking for exceptions

The test below will pass, because a ZeroDivisionError is raised. If a ZeroDivisionError wasn't raised, the test would fail.:

```
from ward import raises, test

@test("a ZeroDivision error is raised when we divide by 0")
def _():
    with raises(ZeroDivisionError):
        1/0
```

If you need to access the exception object that your code raised, you can use with raises(<exc_type>) as <exc_object>:

```
def my_func():
    raise Exception("oh no!")

@test("the message is 'oh no!'")
def _():
    with raises(Exception) as ex:
        my_func()
    assert str(ex.raised) == "oh no!"
```

Note that ex is only populated after the context manager exits, so be careful with your indentation.

3.1.6 Testing *async* code

You can declare any test or fixture as async in order to test asynchronous code:

```
@fixture
async def post():
    return await create_post("hello world")

@test("a newly created post has no children")
async def _(p=post):
    children = await p.children
    assert children == []

@test("a newly created post has an id > 0")
def _(p=post):
    assert p.id > 0
```

3.1.7 Skipping a test

Use the `@skip` decorator to tell Ward not to execute a test.:

```
from ward import skip

@skip
@test("I will be skipped!")
def _():
    # ...
```

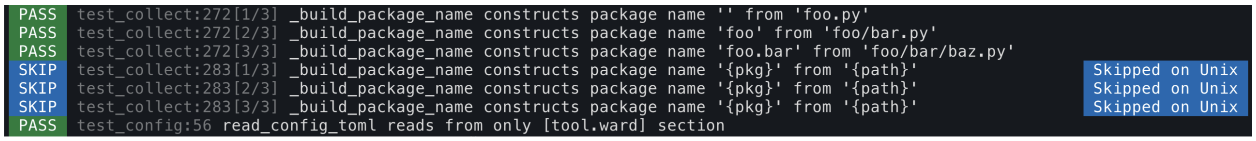
You can pass a `reason` to the `skip` decorator, and it will be printed next to the test name/description during the run

```
@skip("not implemented yet")
@test("everything is okay")
def _():
    # ...
```

To conditionally skip a test in some circumstances (for example, on specific OS's), you can supply a `when` predicate to the `@skip` decorator. This can be either a boolean or a Callable, and will be evaluated just before the test is scheduled to be executed. If it evaluates to `True`, the test will be skipped. Otherwise the test will run as normal.

Here's an example of a test that is skipped on Windows:

```
import platform

@skip("Skipped on Windows", when=platform.system() == "Windows")
@test("_build_package_name constructs package name '{pkg}' from '{path}'")
def _(
    pkg=each("", "foo", "foo.bar"),
    path=each("foo.py", "foo/bar.py", "foo/bar/baz.py"),
):
    m = ModuleType(name="")
    m.__file__ = path
    assert _build_package_name(m) == pkg
```


```
 PASS  test_collect:272[1/3] _build_package_name constructs package name '' from 'foo.py'
 PASS  test_collect:272[2/3] _build_package_name constructs package name 'foo' from 'foo/bar.py'
 PASS  test_collect:272[3/3] _build_package_name constructs package name 'foo.bar' from 'foo/bar/baz.py'
 SKIP  test_collect:283[1/3] _build_package_name constructs package name '{pkg}' from '{path}'          Skipped on Unix
 SKIP  test_collect:283[2/3] _build_package_name constructs package name '{pkg}' from '{path}'          Skipped on Unix
 SKIP  test_collect:283[3/3] _build_package_name constructs package name '{pkg}' from '{path}'          Skipped on Unix
 PASS  test_config:56 read_config_toml reads from only [tool.ward] section
```

3.1.8 Expecting a test to fail

You can mark a test that you expect to fail with the `@xfail` decorator.

```
from ward import xfail

@xfail("its really not okay")
@test("everything is okay")
def _():
    # ...
```

If a test decorated with `@xfail` *does* indeed fail as we expected, it is shown in the results as an `XFAIL`.

You can conditionally apply `@xfail` using the same approach as we described for `@skip` above.

For example, we expect the test below to fail, but *only* when it's run in a Python 3.6 environment.

```
from ward import xfail

@xfail("expected fail on Python 3.6", when=platform.python_version().startswith("3.6"))
@test("everything is okay")
def _():
    # ...
```

If a test marked with this decorator passes unexpectedly, it is known as an XPASS (an unexpected pass).

If an XPASS occurs during a run, the run will be considered a failure.

3.2 Running Tests

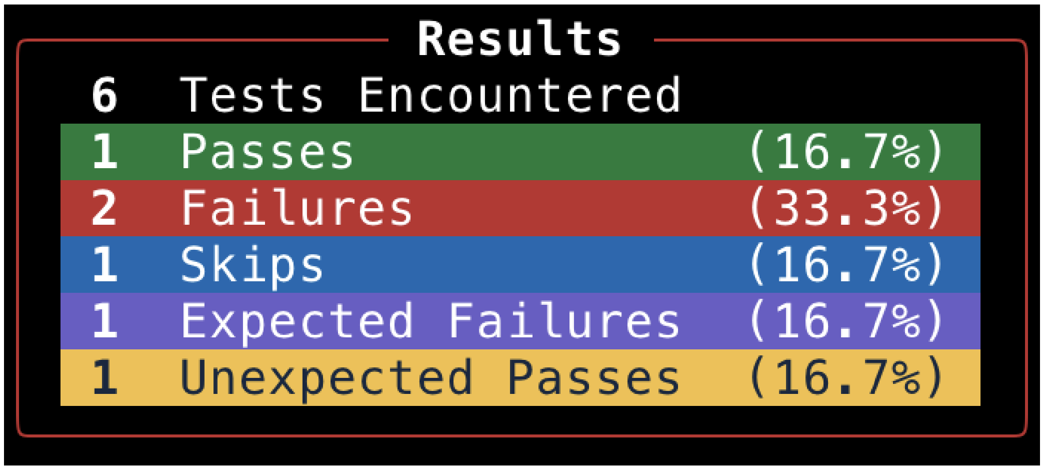
To find and run tests in your project, you can run ward without any arguments.

This will recursively search through the current directory for modules with a name starting with test_ or ending with _test, and execute any tests contained in the modules it finds.

3.2.1 Test outcomes

A test in Ward can finish with one of several different outcomes. The outcome of a test will be clearly indicated during the run, and a summary of those outcomes is displayed after the run completes or is cancelled.

- PASS: The test passed. It completed without any exceptions occurring or assertions failing.
- FAIL: The test failed. An exception occurred or an assertion failed.
- SKIP: The test was skipped. It wasn't executed at all because it was decorated with @skip.
- XFAIL: An expected failure. The test is decorated with @xfail, indicating that we currently expect it to fail… and it did!
- XPASS: An unexpected pass. The test is decorated with @xfail, indicating that we expected it to fail. However, the test passed unexpectedly.
- DRYRUN: The status is only used during a dry-run (using the --dry-run option). The test nor any injected fixtures were executed.



```
                Results
  6  Tests Encountered
  1  Passes                (16.7%)
  2  Failures              (33.3%)
  1  Skips                 (16.7%)
  1  Expected Failures     (16.7%)
  1  Unexpected Passes     (16.7%)
```

3.2.2 Specifying test paths with `--path`

You can run tests in a specific directory or module using the `--path` option. For example, to run all tests inside a directory named `tests`: `ward --path tests`

To run tests in the current directory, you can just type `ward`, which is functionally equivalent to `ward --path .`.

You can directly specify a test module, for example: `ward --path tests/api/test_get_user.py`.

You can supply multiple test directories by providing the `--path` option multiple times: `ward --path "unit" --path "integration"`.

Ward will run all tests it finds across all given paths. If one of the specified paths is contained within another, they'll only be included once. Ward will only run a test once per session.

3.2.3 Excluding modules or paths with `--exclude`

`ward --exclude glob1 --exclude glob2`

You can tell Ward to ignore specific modules or directories using the `--exclude` command line option. This option can be supplied multiple times, and supports glob patterns.

You can also exclude paths using `pyproject.toml`:

```
[tool.ward]
exclude = ["glob1", "glob2"]
```

3.2.4 Selecting tagged tests with `--tags`

You can select which tests to run based on a "test expressions" using the `--tags` option: `ward --tags EXPR`.

A tag expression is an infix boolean expression that can be used to accurately select a subset of tests you wish to execute. Tests are tagged using the tags keyword argument of the `@test` decorator (e.g. `@test("eggs are green", tags=["unit"])`.)

For example, if you wanted to run all tests tagged with either `android` or `ios`, run `ward --tags "android or ios"`.

Here are some examples of tag expressions and what they mean:

- `slow`: tests tagged with `slow`
- `unit and integration`: tests tagged with both `unit` and `integration`
- `big and not slow`: tests tagged with `big` that aren't also tagged with `slow`
- `android or ios`: tests tagged with either `android` or `ios`

You can use parentheses in tag expressions to change the precedence rules to suit your needs.

3.2.5 Loosely search for tests with `--search`

You can choose to limit which tests are collected and ran by Ward using the `--search` option. Module names, test descriptions and test function bodies will be searched, and those which contain the argument will be ran.

Here are some examples:

- Run all tests that call the `fetch_users` function: `ward --search "fetch_users("`
- Run all tests that check if a `ZeroDivisionError` is raised: `ward --search "raises(ZeroDivisionError)"`
- Run all tests decorated with the `@xfail` decorator: `ward --search "@xfail"`
- Run a test described with `"my_function should return False"`: `ward --search "my_function should return False"`
- Running tests inside a module: The search takes place on the fully qualified name, so you can run a single module (e.g. my_module) using the following command: `ward --search my_module.`

Of course, if a test name or body contains the string `"my_module."`, that test will also be selected and will run.

This approach is useful for quickly running tests which match a simple query, making it useful for development.

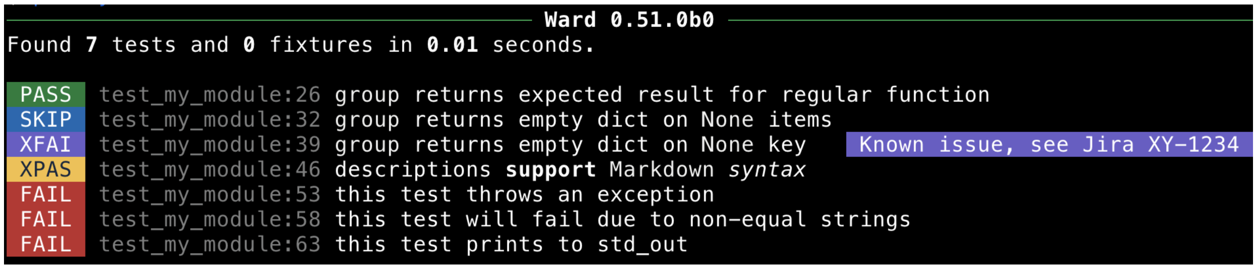
3.2.6 Customising the output with `--test-output-style`

As your project grows, it may become impractical to print each test result on its own line. Ward provides alternative test output styles that can be configured using the `--test-output-style` option.

```
ward --test-output-style [test-per-line|dots-module|dots-global]
```

`test-per-line` (default)

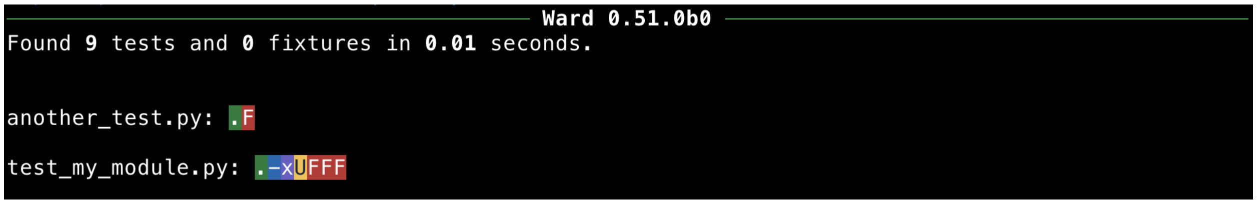
The default test output of Ward looks like this (`--test-output-style=test-per-line`):



```
───────────────────────────────────────────── Ward 0.51.0b0 ─────────────────────────────────
Found 7 tests and 0 fixtures in 0.01 seconds.

 PASS  test_my_module:26 group returns expected result for regular function
 SKIP  test_my_module:32 group returns empty dict on None items
 XFAI  test_my_module:39 group returns empty dict on None key    Known issue, see Jira XY-1234
 XPAS  test_my_module:46 descriptions support Markdown syntax
 FAIL  test_my_module:53 this test throws an exception
 FAIL  test_my_module:58 this test will fail due to non-equal strings
 FAIL  test_my_module:63 this test prints to std_out
```

`dots-module`

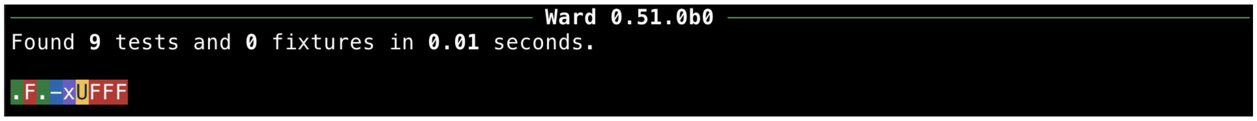
If you run Ward with `--test-output-style=dots-module`, each module will be printed on its own line, and a single character will be used to represent the outcome of each test in that module:



```
───────────────────────────────────────────── Ward 0.51.0b0 ─────────────────────────────────
Found 9 tests and 0 fixtures in 0.01 seconds.


another_test.py: .F

test_my_module.py: .-xUFFF
```

`dots-global`

If that is still too verbose, you may wish to represent every test outcome with a single character, without grouping them by modules (`--test-output-style=dots-global`):



```
──────────────────────────────── Ward 0.51.0b0 ────────────────────────────────
Found 9 tests and 0 fixtures in 0.01 seconds.

.F.-xUFFF
```

3.2.7 Displaying test session progress with `--progress-style`

Ward offers two ways of informing you of progress through a test run: inline progress percentage (on by default), and/or a dynamic progress bar.

By default, the percentage progress through a test run will appear at the right hand side of the output, which corresponds to `--progress-style inline`.

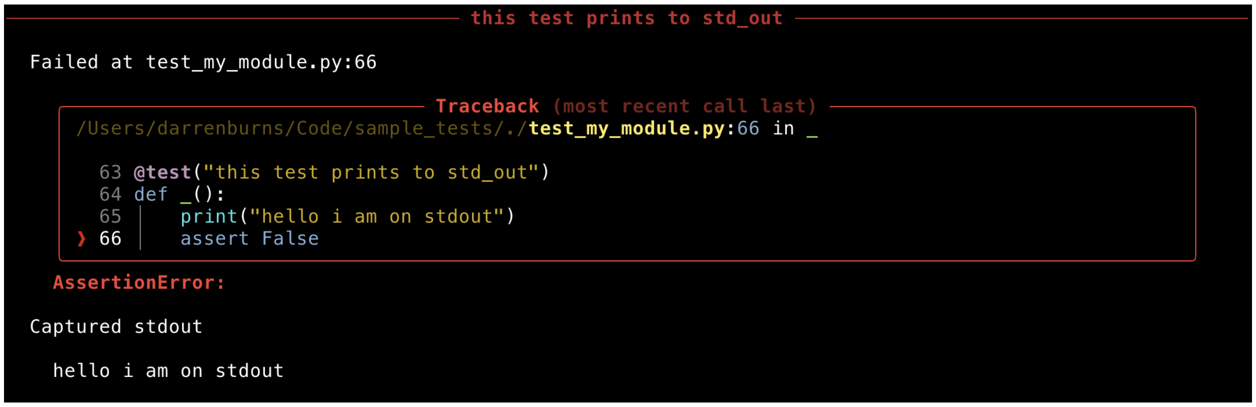
You can also have Ward display a dynamic progress bar during the test run, using the `--progress-style bar` option.

If you wish, can pass supply `--progress-style` with multiple times (to display a progress bar and inline progress, for example).

Warning: The progress bar is currently only available with the default output mode (`--test-output-style test-per-line`).

3.2.8 Output capturing

By default, Ward captures everything that is written to stdout and stderr as your tests run. If a test fails, everything that was printed during the time it was running will be printed as part of the failure output.



```
─────────────────────────── this test prints to std_out ───────────────────────────

 Failed at test_my_module.py:66

    ┌─────────────────────── Traceback (most recent call last) ───────────────────────┐
    │ /Users/darrenburns/Code/sample_tests/./test_my_module.py:66 in _               │
    │                                                                                 │
    │   63 @test("this test prints to std_out")                                       │
    │   64 def _():                                                                   │
    │   65 │    print("hello i am on stdout")                                         │
    │ ❱ 66 │    assert False                                                          │
    └─────────────────────────────────────────────────────────────────────────────────┘
   AssertionError:

 Captured stdout

   hello i am on stdout
```

With output capturing enabled, if you run a debugger such as pdb during test execution, everything it writes to the stdout will be captured by Ward too.

Disabling output capturing with `--no-capture-output`

If you wish to disable output capturing you can do so using the `--no-capture-output` flag on the command line. Anything printed to stdout or stderr will no longer be captured by Ward, and will be printed to the terminal as your tests run, regardless of outcome.

You can also disable output capturing using the `capture-output` config in your `pyproject.toml`:

```toml
[tool.ward]
capture-output = false
```

3.2.9 Randomise test execution order with `--order random`

Use `--order "random"` when running your tests to have Ward randomise the order they run in: `ward --order "random"`.

Running tests in a random order can help identify tests that have hidden dependencies on each other. Tests should pass regardless of the order they run in, and they should pass if run in isolation.

To have Ward always run tests in a random order, use the `order` config in your `pyproject.toml`:

```toml
[tool.ward]
order = "random"
```

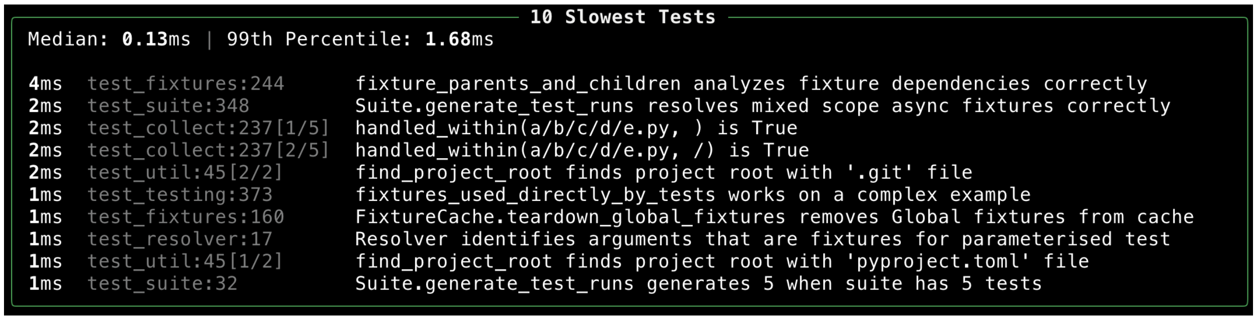
3.2.10 Cancelling after a number of failures with `--fail-limit`

If you wish for Ward to cancel a run immediately after a specific number of failing tests, you can use the `--fail-limit` option. To have a run end immediately after 5 tests fail:

`ward --fail-limit 5`

3.2.11 Finding slow running tests with `--show-slowest`

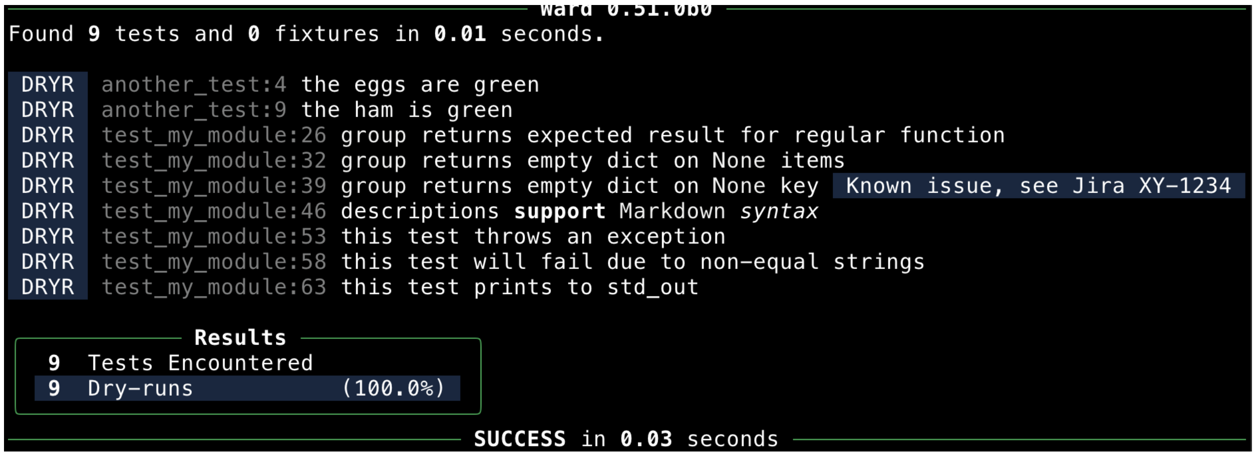
Use `--show-slowest N` to print the N tests with the highest execution time after the test run completes.



```
                                   10 Slowest Tests
Median: 0.13ms | 99th Percentile: 1.68ms

4ms  test_fixtures:244      fixture_parents_and_children analyzes fixture dependencies correctly
2ms  test_suite:348         Suite.generate_test_runs resolves mixed scope async fixtures correctly
2ms  test_collect:237[1/5]  handled_within(a/b/c/d/e.py, ) is True
2ms  test_collect:237[2/5]  handled_within(a/b/c/d/e.py, /) is True
2ms  test_util:45[2/2]      find_project_root finds project root with '.git' file
1ms  test_testing:373       fixtures_used_directly_by_tests works on a complex example
1ms  test_fixtures:160      FixtureCache.teardown_global_fixtures removes Global fixtures from cache
1ms  test_resolver:17       Resolver identifies arguments that are fixtures for parameterised test
1ms  test_util:45[1/2]      find_project_root finds project root with 'pyproject.toml' file
1ms  test_suite:32          Suite.generate_test_runs generates 5 when suite has 5 tests
```

3.2.12 Performing a dry run with `--dry-run`

Use the `--dry-run` option to have Ward search for and collect tests without running them (or any fixtures they depend on). When using `--dry-run`, tests will return with an outcome of `DRYRUN`.



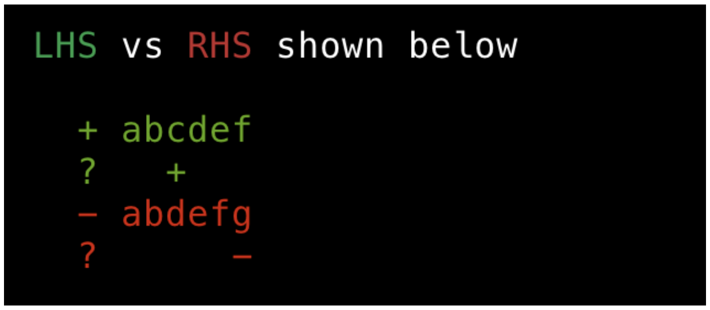
```
──────────────────────────── ward 0.51.0b0 ────────────────────────────
Found 9 tests and 0 fixtures in 0.01 seconds.

 DRYR  another_test:4 the eggs are green
 DRYR  another_test:9 the ham is green
 DRYR  test_my_module:26 group returns expected result for regular function
 DRYR  test_my_module:32 group returns empty dict on None items
 DRYR  test_my_module:39 group returns empty dict on None key  Known issue, see Jira XY-1234
 DRYR  test_my_module:46 descriptions support Markdown syntax
 DRYR  test_my_module:53 this test throws an exception
 DRYR  test_my_module:58 this test will fail due to non-equal strings
 DRYR  test_my_module:63 this test prints to std_out

 ┌────────── Results ──────────┐
 │  9  Tests Encountered        │
 │  9  Dry-runs        (100.0%) │
 └──────────────────────────────┘
──────────────────────── SUCCESS in 0.03 seconds ────────────────────────
```

This is useful for determining which tests Ward would run if invoked normally.

Format strings in test descriptions may not be resolved during a dry-run, since no fixtures are evaluated and the data may therefore be missing.

3.2.13 Displaying symbols in diffs with `--show-diff-symbols`

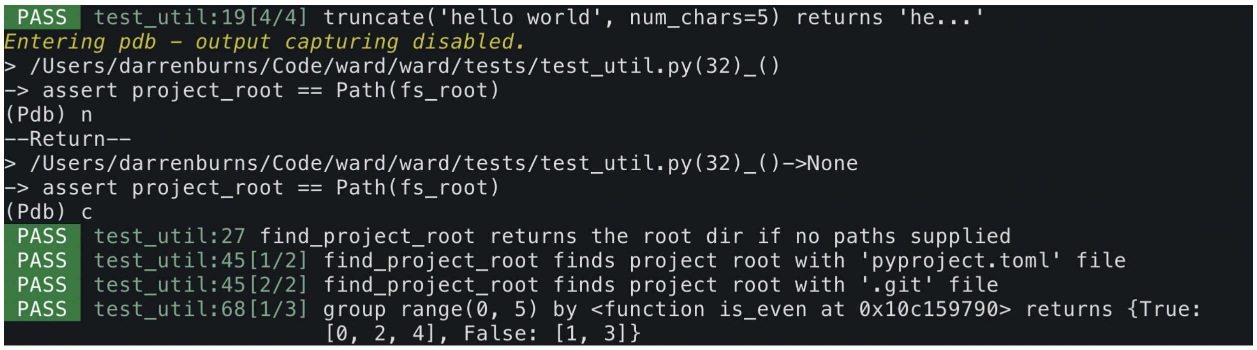
Use `--show-diff-symbols` when invoking `ward` in order to have the diff output present itself with symbols instead of the colour-based highlighting. This may be useful in a continuous integration environment that doesn't support coloured terminal output.



```
LHS vs RHS shown below

  + abcdef
  ?    +
  - abdefg
  ?      -
```

3.2.14 Debugging your code with `pdb/breakpoint()`

Ward will automatically disable output capturing when you use *pdb.set_trace()* or *breakpoint()*, and re-enable it when you exit the debugger.



```
 PASS  test_util:19[4/4] truncate('hello world', num_chars=5) returns 'he...'
Entering pdb - output capturing disabled.
> /Users/darrenburns/Code/ward/ward/tests/test_util.py(32)_()
-> assert project_root == Path(fs_root)
(Pdb) n
--Return--
> /Users/darrenburns/Code/ward/ward/tests/test_util.py(32)_()->None
-> assert project_root == Path(fs_root)
(Pdb) c
 PASS  test_util:27 find_project_root returns the root dir if no paths supplied
 PASS  test_util:45[1/2] find_project_root finds project root with 'pyproject.toml' file
 PASS  test_util:45[2/2] find_project_root finds project root with '.git' file
 PASS  test_util:68[1/3] group range(0, 5) by <function is_even at 0x10c159790> returns {True:
                         [0, 2, 4], False: [1, 3]}
```

3.3 Fixtures

A *fixture* is a function that provides tests with the data they need in order to run.

They provide a modular, composable alternative to `setup/before*` and `teardown/after*` methods that appear in many test frameworks.

3.3.1 Declaring and using a simple fixture

We can declare a fixture using the `@fixture` decorator. Let's define a fixture that represents a user on a website.

```python
from ward import fixture

@fixture
def user():
    return User(id=1, name="sam")
```

Now lets add a test that will make use of the user fixture.

```python
from ward import test

@test("fetch_user_by_id should return the expected User object")
def _(expected_user=user):
    fetched_user = fetch_user_by_id(id=expected_user.id)
    assert fetched_user == expected_user
```

By directly binding the fixture as a default argument to our test function, we've told Ward to resolve the fixture and inject it into our test. Inside our test, the variable `expected_user` is the object `User(id=1, name="sam")`.

3.3.2 The `@using` decorator

An alternative approach to injecting fixtures into tests is the `@using` decorator.

This approach lets us use positional arguments in our test signature, and declare which fixture each argument refers to using the decorator.

Here's how we'd inject our user fixture into a test with using:

```python
from ward import expect, test, using

@test("fetch_user_by_id should return the expected User object")
@using(expected_user=user)
def _(expected_user):
    fetched_user = fetch_user_by_id(id=expected_user.id)
    assert fetched_user == expected_user
```

In the example above, we tell Ward to bind the resolved value of the user fixture to the expected_user position argument.

3.3.3 Fixture scope

By default, a fixture is executed immediately before each test it is injected into.

If the code inside your fixtures is expensive to execute, it may not be practical to have it run before every test that depends on it.

To solve this problem, Ward lets you give a *"scope"* to your fixtures. The scope of a fixture determines how long it is cached for.

Ward supports 3 scopes: `test` (default), `module`, and `global`.

- A test scoped fixture will be evaluated at most once per test.
- A module scoped fixture will be evaluated at most once per module.
- A global scoped fixture will be evaluated at most once per invocation of ward.

To make the user fixture global scope, we can change the decorator call to `@fixture(scope=Scope.Global)`.

```python
from ward import fixture, Scope

@fixture(scope=Scope.Global)  # @fixture(scope="global") also works
def user():
    return User(id=1, name="sam")
```

This fixture will be executed and cached the first time it is injected into a test.

Because it has a global scope, Ward will pass the cached value into all other tests that use it.

If user instead had a scope of `Scope.Module`, then Ward would re-evaluate the fixture when it's required by a test in any other module.

Careful management of fixture scope can drastically reduce the time and resources required to run a suite of tests.

As a general rule of thumb, if the value returned by a fixture is immutable, or we know that no test will mutate it, then we can make it global.

Warning: You should never mutate a global or module scoped fixture. Doing so breaks the isolated nature of tests, and introduces hidden dependencies between them. Ward will warn you if it detects a global or module scoped fixture has been mutated inside a test (coming in v1.0).

3.3.4 Fixture composition

Fixtures can be composed by injecting them into each other.

You can inject a fixture into another fixture in the same way that you'd inject it into a test: by binding it as a default argument.

```
@fixture
def name():
    return "sam"

@fixture
def user(name=name):
    return {"name": name}

@test("fixtures can be composed")
def _(name=name, user=user):
    assert user["name"] == name
```

In the example above, user depends on name, and the test depends on both user and name. Both fixtures are test scoped, so they are evaluated at most once per test. This means that the name instance that Ward passes into user is the same instance it passes into the test.

```
PASS test_composition:14: fixtures can be composed
```

3.3.5 Running teardown code

Fixtures have the ability to cleanup after themselves.

For a fixture to run teardown code, it must be declared as a *generator function*.

Notice how we `yield` the value of the fixture in the test below. Ward will inject the yielded value into the test, and after the test has run, all code below the `yield` will be executed.

```
from ward import test, fixture

@fixture
def database():
    print("1. I'm setting up the database!")
    db_conn = setup_database()
    yield db_conn
    db_conn.close()
    print("3. I've torn down the database!")

@test(f"Bob is one of the users contained in the database")
def _(db=database):
    print("2. I'm running the test!")
    users = get_all_users(db)
    assert "Bob" in users
```

The output captured by Ward whilst the test above runs is:

1. I'm setting up the database!
2. I'm running the test!

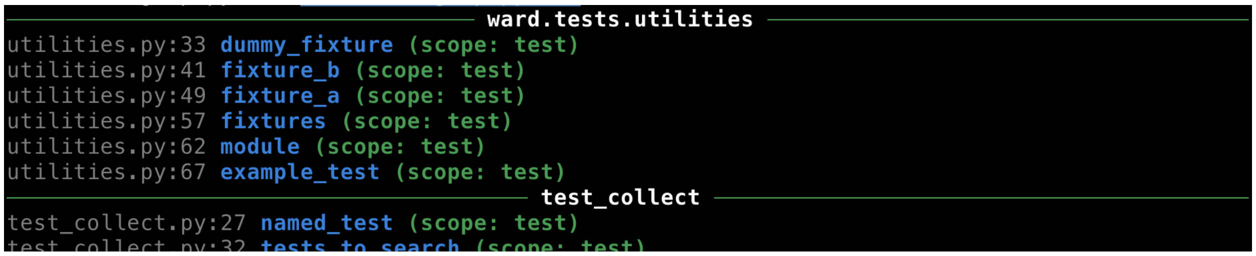
3. I've torn down the database!

Global and module scoped fixtures can also contain teardown code:

- In the case of a module scoped fixture, the teardown code will run after the test module completes.
- In the case of a global scoped fixture, the teardown code will run after the whole test suite completes.
- If an exception occurs during the setup phase of the fixture, the teardown phase will not run.
- If an exception occurs during the running of a test, the teardown phase of any fixtures that that test depends on will run.

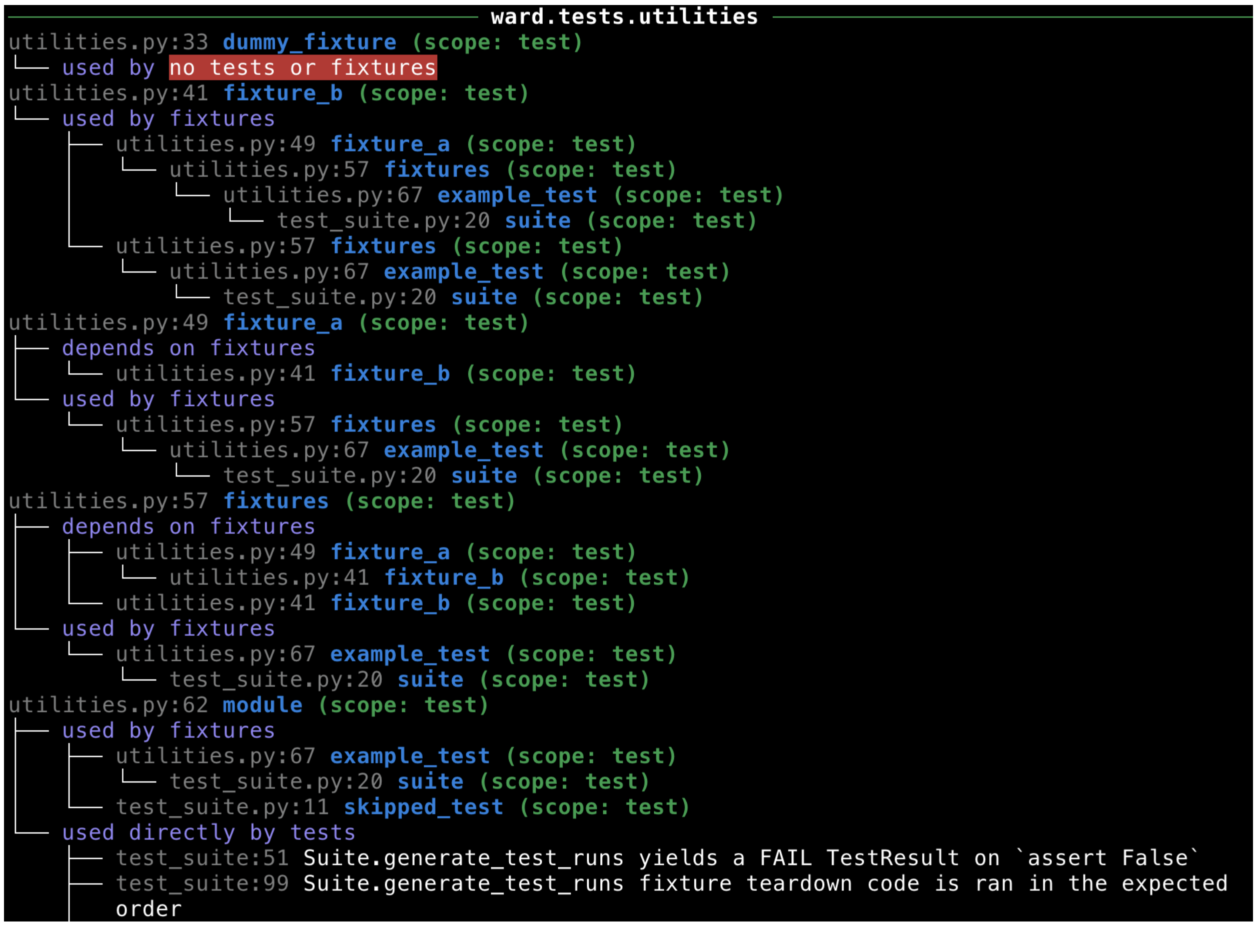
3.3.6 Inspecting fixtures

You can view all of the fixtures in your project using the `ward fixtures` command.



```
──────────────────────────────────── ward.tests.utilities ──────────────────────────────────
utilities.py:33 dummy_fixture (scope: test)
utilities.py:41 fixture_b (scope: test)
utilities.py:49 fixture_a (scope: test)
utilities.py:57 fixtures (scope: test)
utilities.py:62 module (scope: test)
utilities.py:67 example_test (scope: test)
──────────────────────────────────────── test_collect ──────────────────────────────────────
test_collect.py:27 named_test (scope: test)
test_collect.py:32 tests_to_search (scope: test)
```

To view the dependency graph of fixtures, and detect fixtures that are unused, you can run `ward fixtures --show-dependency-trees`:



```
──────────────────────── ward.tests.utilities ────────────────────────
utilities.py:33 dummy_fixture (scope: test)
└── used by no tests or fixtures
utilities.py:41 fixture_b (scope: test)
└── used by fixtures
    ├── utilities.py:49 fixture_a (scope: test)
    │   └── utilities.py:57 fixtures (scope: test)
    │       └── utilities.py:67 example_test (scope: test)
    │           └── test_suite.py:20 suite (scope: test)
    └── utilities.py:57 fixtures (scope: test)
        └── utilities.py:67 example_test (scope: test)
            └── test_suite.py:20 suite (scope: test)
utilities.py:49 fixture_a (scope: test)
├── depends on fixtures
│   └── utilities.py:41 fixture_b (scope: test)
└── used by fixtures
    └── utilities.py:57 fixtures (scope: test)
        └── utilities.py:67 example_test (scope: test)
            └── test_suite.py:20 suite (scope: test)
utilities.py:57 fixtures (scope: test)
├── depends on fixtures
│   ├── utilities.py:49 fixture_a (scope: test)
│   │   └── utilities.py:41 fixture_b (scope: test)
│   └── utilities.py:41 fixture_b (scope: test)
└── used by fixtures
    └── utilities.py:67 example_test (scope: test)
        └── test_suite.py:20 suite (scope: test)
utilities.py:62 module (scope: test)
├── used by fixtures
│   ├── utilities.py:67 example_test (scope: test)
│   │   └── test_suite.py:20 suite (scope: test)
│   └── test_suite.py:11 skipped_test (scope: test)
└── used directly by tests
    ├── test_suite:51 Suite.generate_test_runs yields a FAIL TestResult on `assert False`
    ├── test_suite:99 Suite.generate_test_runs fixture teardown code is ran in the expected
    │   order
```

3.4 Configuration

3.4.1 How does Ward use `pyproject.toml`?

You can configure Ward using the standard `pyproject.toml` configuration file, defined in PEP 518.

You don't need a `pyproject.toml` file to use Ward.

If you do decide to use one, Ward will find and read your `pyproject.toml` file, and treat the values inside it as defaults.

If you pass an option via the command line that also appears in your `pyproject.toml`, the option supplied via the command line takes priority.

3.4.2 Where does Ward look for `pyproject.toml`?

The algorithm Ward uses to discover your `pyproject.toml` is described at a high level below.

1. Find the common base directory of all files passed in via the `--path` option (default to the current working directory).
2. Starting from this directory, look at all parent directories, and return the file if it is found.
3. If a directory contains a `.git` directory/file, a .hg directory, or the `pyproject.toml` file, stop searching.

This is the same process Black (the popular code formatting tool) uses to discover the file.

3.4.3 Example `pyproject.toml` config file

The `pyproject.toml` file contains different sections for different tools. Ward uses the `[tool.ward]` section, so all of your Ward configuration should appear there:

```toml
[tool.ward]
path = ["unit_tests", "integration_tests"]  # supply multiple paths using a list
capture-output = false  # enable or disable output capturing (e.g. to use debugger)
order = "standard"  # or 'random'
output-mode = "test-per-line"  # or 'dots-global', 'dot-module'
fail-limit = 20  # stop the run if 20 fails occur
search = "my_function"  # search in test body or description
progress-style = ["bar"]  # display a progress bar during the run
```

3.5 Your First Tests

In this tutorial, we'll write two tests using Ward. These tests aren't realistic, nor is the function we're testing. This page exists to give a tour of some of the main features of Ward and their motivations. We'll define reusable test data in a fixture, and pass that data into our tests. Finally, we'll look at how we can cache that test data to improve performance.

3.5.1 Installing Ward

Ward is available on PyPI, so it can be installed using pip: `pip install ward`.

When you run `ward` with no arguments, it will recursively look for tests starting from your current directory.

Ward will look for tests in any Python module with a name that starts with `test_` or ends with `_test`.

We're going to write tests for a function called `contains`:

```python
def contains(list_of_items, target_item)
```

This function should return `True` if the `target_item` is contained within `list_of_items`. Otherwise it should return `False`.

3.5.2 Our first test

Tests in Ward are just Python functions decorated with `@test(description: str)`.

Functions with this decorator can be named `_`. We'll tell readers what the test does using a plain English description rather than the function name.

Our test is contained within a file called `test_contains.py`:

```
from contains import contains
from ward import test

@test("contains returns True when the item is in the list")
def _():
    list_of_ints = list(range(100000))
    result = contains(list_of_ints, 5)
    assert result
```

In this file, we've defined a single test function called `_`. It's been decorated with `@test`, and has a helpful description. We don't have to read the code inside the test to understand its purpose.

The description can be queried when running a subset of tests. You may decide to use your own conventions inside the description in order to make your tests highly queryable.

Now we can run `ward` in our terminal.

Ward will find and run the test, and confirm that the test PASSED with a message like the one below.

```
PASS test_contains:4: contains returns True when item is in list
```

3.5.3 Fixtures: Extracting common setup code

Lets add another test.

```
@test("contains returns False when item is not in list")
def _():
    list_of_ints = list(range(100000))
    result = contains(list_of_ints, -1)
    assert not result
```

This test begins by instantiating the same list of 10 integers as the first test. This duplicated setup code can be extracted out into a fixture so that we don't have to repeat ourselves at the start of every test.

The `@fixture` decorator lets us define a fixture, which is a unit of test setup code. It can optionally contain some additional code to clean up any resources the it used (e.g. cleaning up a test database).

Lets define a fixture immediately above the tests we just wrote.

```
from ward import fixture

@fixture
def list_of_ints():
    return list(range(100000))
```

We can now rewrite our tests to make use of this fixture. Here's how we'd rewrite the second test.

```
@test("contains returns False when item is not in list")
def _(l=list_of_ints):
    result = contains(l, -1)
    assert not result
```

By binding the name of the fixture as a default argument to the test, Ward will resolve it before the test runs, and inject it into the test.

By default, a fixture is executed immediately before being injected into a test. In the case of `list_of_ints`, that could be problematic if lots of tests depend on it. Do we really want to instantiate a list of 100000 integers before each of those tests? Probably not.

3.5.4 Improving performance with fixture scoping

To avoid this repeated expensive test setup, you can tell Ward what the scope of a fixture is. The scope of a fixture defines how long it should be cached for.

Ward supports 3 scopes: test (default), module, and global.

- A *test* scoped fixture will be evaluated at most once per test.
- A *module* scoped fixture will be evaluated at most once per test module.
- A *global* scoped fixture will be evaluated at most once per invocation of `ward`.

If a fixture is never injected into a test or another fixture, it will never be evaluated.

We can safely say that we only need to generate our `list_of_ints` once, and we can reuse its value in every test that depends on it. So lets give it a global scope:

```
from ward import fixture, Scope

@fixture(scope=Scope.Global)  # or scope="global"
def list_of_ints():
    return list(range(100000))
```

With this change, our fixture will now only be evaluated once, regardless of how many tests depend on it. Careful management of fixture scope can drastically reduce the time and resources required to run a suite of tests.

As a general rule of thumb, if the value returned by a fixture is immutable, or we know that no test will mutate it, then we can make it global.

Warning: You should *never* mutate a global or module scoped fixture. Doing so breaks the isolated nature of tests, and introduces hidden dependencies between them.

3.5.5 Summary

In this tutorial, you learned how to write your first tests with Ward. We covered how to write a test, inject a fixture into it, and cache the fixture for performance.

3.6 Testing a Flask App

Let's write a couple of tests using Ward for the following Flask application (`app.py`).

It's an app that contains a single endpoint. If you run this app with `python -m app`, then visit `localhost:5000/users/alice` in your browser, you should see that the application returns the response `The user is alice`.

```python
# file: app.py

from flask import Flask

app = Flask(__name__)

@app.route("/users/<string:username>")
def get_user(username: str):
    return f"The user is {username}"

if __name__ == "__main__":
    app.run()
```

A common way of testing Flask applications is to use the helpful `TestClient` class. Using `TestClient`, we can easily make requests to our app, and see how it behaves and responds.

Before going any further, let's install `ward` and `flask`:

```
pip install ward flask
```

Create a new file called `test_app.py`, and inside it, let's define a fixture to configure the Flask application for testing. We'll inject this fixture into each of our tests, and this will allow us to send requests to our application and ensure it's behaving correctly!

```python
from ward import fixture
from app import app

@fixture(scope="global")
def test_client():
    app.config['TESTING'] = True  # For better error reports
    with app.test_client() as client:
        yield client
```

This fixture yields an instance of the `TestClient`, which can be accessed from the Flask object we used to create our app.

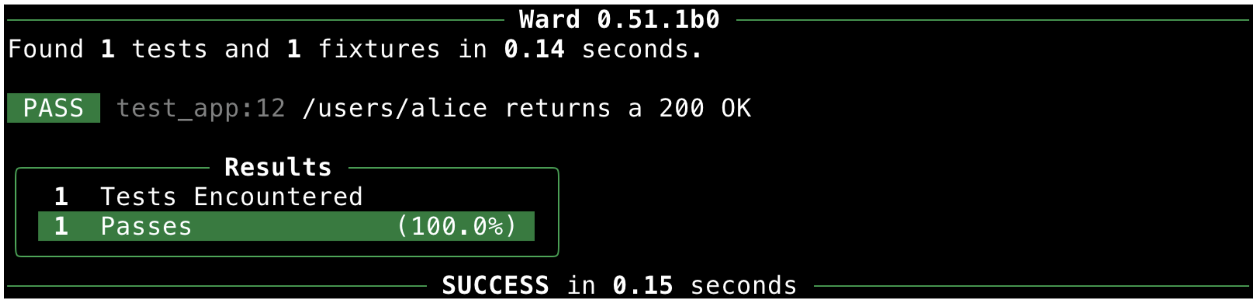
We only need to create a single test client, which we can reuse across all tests in our test session, so the `scope` of the fixture is set to `"global"`.

Yielding the client from within the with statement means that any resources used by the client will be cleaned up after the test session completes.

Now we'll create our first test, which will check that our app returns the correct HTTP status code when we visit our endpoint with a valid username ("alice"). The status code we expect to see in this case is an `HTTP 200 (OK)`.

```
from ward import fixture, test
from app import app

@fixture(scope="global")
def test_client():
    app.config['TESTING'] = True  # For better error reports
    with app.test_client() as client:
        yield client

@test("/users/alice returns an 200 OK")
def _(client=test_client):
    res = client.get("/users/alice")
    assert res.status_code == 200
```

We can run our test by running ward in our terminal:



```
──────────────────────────── Ward 0.51.1b0 ────────────────────────────
Found 1 tests and 1 fixtures in 0.14 seconds.

 PASS  test_app:12 /users/alice returns a 200 OK

 ─────────── Results ───────────
   1  Tests Encountered
   1  Passes                (100.0%)

────────────────────── SUCCESS in 0.15 seconds ──────────────────────
```

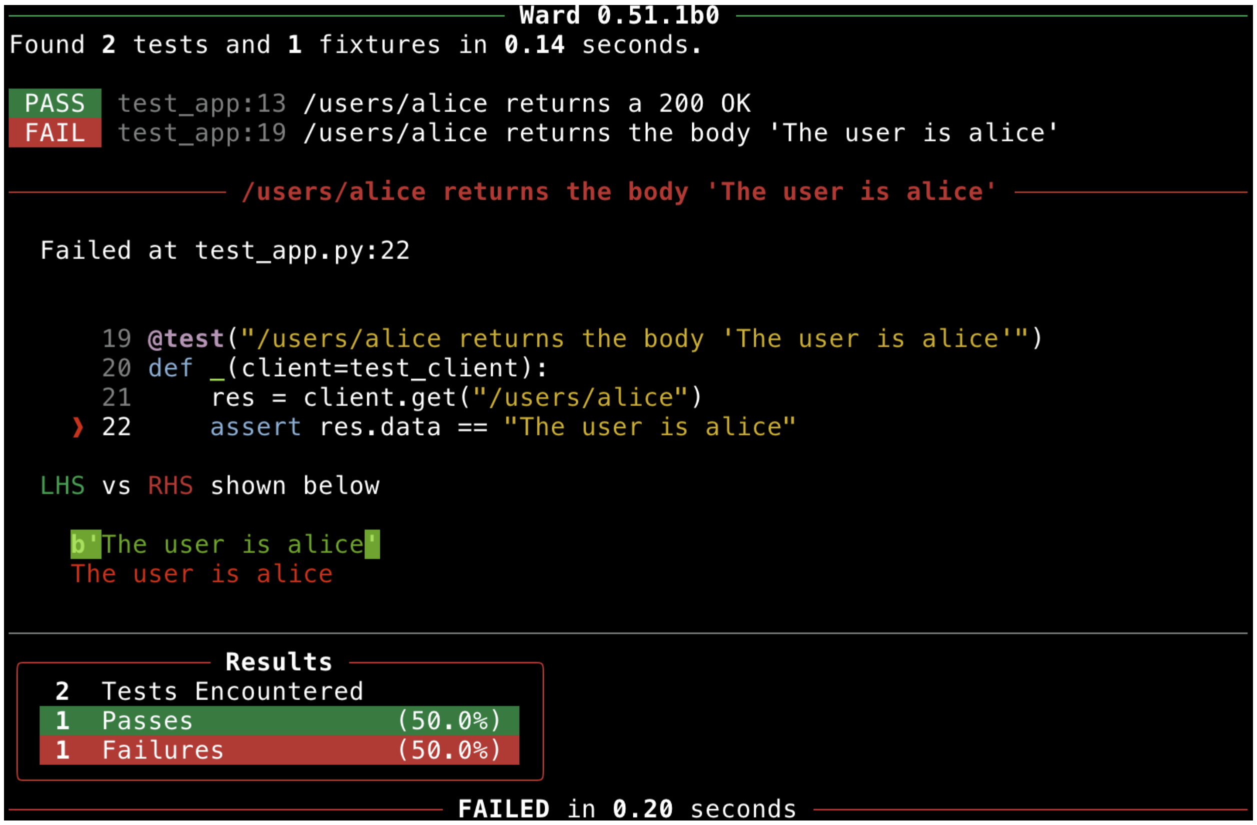
Success! It's a PASS! The fully green bar indicates a 100% success rate!

Tip: If we had lots of other, unrelated endpoints in our API and we only wanted to run the tests that affect the `/users/` endpoint, we could do so using the command `ward --search "/users/"`.

Let's add another test below, to check that the body of the response is what we expect it to be.

```
@test("/users/alice returns the body 'The user is alice'")
def _(client=test_client):
    res = client.get("/users/alice")
    assert res.data == "The user is alice"
```

Running our tests again, we see that our new test fails!



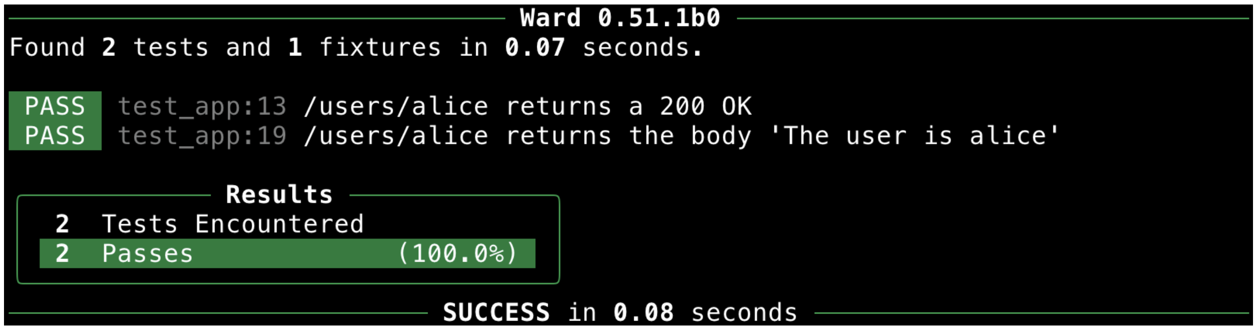
```
─────────────────────────────── Ward 0.51.1b0 ───────────────────────────────
Found 2 tests and 1 fixtures in 0.14 seconds.

 PASS  test_app:13 /users/alice returns a 200 OK
 FAIL  test_app:19 /users/alice returns the body 'The user is alice'

────────── /users/alice returns the body 'The user is alice' ──────────

  Failed at test_app.py:22


    19 @test("/users/alice returns the body 'The user is alice'")
    20 def _(client=test_client):
    21     res = client.get("/users/alice")
  ❯ 22     assert res.data == "The user is alice"

  LHS vs RHS shown below

    b'The user is alice'
    The user is alice

─────────────────────────────────────────────────────────────────────────────
 ┌────── Results ───────────────┐
 │ 2  Tests Encountered         │
 │ 1  Passes             (50.0%)│
 │ 1  Failures           (50.0%)│
 └──────────────────────────────┘
──────────────────────── FAILED in 0.20 seconds ────────────────────────
```

Looking at our output, we can see that while we expected the output to be The user is alice, it was actually `b'The user is alice'`. Ward highlights the specific differences between the expected value and the actual value to help you quickly spot bugs.

This test failed because because `res.data` returns a `bytes` object, not a string like our we thought when we wrote our test. Let's correct the test:

```python
@test("/users/alice returns the body 'The user is alice'")
def _(client=test_client):
    res = client.get("/users/alice")
    assert res.data == b"The user is alice"
```

If we run our tests again using `ward`, we see that they both PASS!



```
─────────────────────────────── Ward 0.51.1b0 ───────────────────────────────
Found 2 tests and 1 fixtures in 0.07 seconds.

 PASS  test_app:13 /users/alice returns a 200 OK
 PASS  test_app:19 /users/alice returns the body 'The user is alice'

 ┌────── Results ─────────────────┐
 │ 2  Tests Encountered           │
 │ 2  Passes             (100.0%) │
 └────────────────────────────────┘
─────────────────────── SUCCESS in 0.08 seconds ───────────────────────
```

3.7 `ward.testing`

3.7.1 Standard API

3.7.2 Plugin API (in development)

This section contains items from this module that are intended for use by plugin authors or those contributing to Ward itself. If you're just using Ward to write your tests, this section isn't relevant.

3.8 `ward.fixtures`

3.8.1 Standard API

3.8.2 Plugin API (in development)

This section contains items from this module that are intended for use by plugin authors or those contributing to Ward itself. If you're just using Ward to write your tests, this section isn't relevant.