Testing

2024-02-13

Writing and Running Tests

Note: a part of this document refers to functionality provided by the included testing_utils.py, the bulk of which I have developed while I worked at HuggingFace.

This document covers both `pytest` and `unittest` functionalities and shows how both can be used together.

Running tests

Run all tests

```
pytest
```

I use the following alias:

```
alias pyt="pytest --disable-warnings --instafail -rA"
```

which tells pytest to:

- disable warning
- `--instafail` shows failures as they happen, and not at the end
- `-rA` generates a short test summary info

it requires you to install:

`pip install pytest-instafail`

Getting the list of all tests

Show all tests in the test suite:

```
pytest --collect-only -q
```

Show all tests in a given test file:

```
pytest tests/test_optimization.py --collect-only -q
```

I use the following alias:

```
alias pytc="pytest --disable-warnings --collect-only -q"
```

Run a specific test module

To run an individual test module:

```
pytest tests/utils/test_logging.py
```

Run specific tests

If `unittest` is used, to run specific subtests you need to know the name of the `unittest` class containing those tests. For example, it could be:

```
pytest tests/test_optimization.py::OptimizationTest::test_adam_w
```

Here:

- `tests/test_optimization.py` - the file with tests
- `OptimizationTest` - the name of the test class
- `test_adam_w` - the name of the specific test function

If the file contains multiple classes, you can choose to run only tests of a given class. For example:

```
pytest tests/test_optimization.py::OptimizationTest
```

will run all the tests inside that class.

As mentioned earlier you can see what tests are contained inside the `OptimizationTest` class by running:

```
pytest tests/test_optimization.py::OptimizationTest --collect-only -q
```

You can run tests by keyword expressions.

To run only tests whose name contains `adam`:

```
pytest -k adam tests/test_optimization.py
```

Logical `and` and `or` can be used to indicate whether all keywords should match or either. `not` can be used to negate.

To run all tests except those whose name contains `adam`:

```
pytest -k "not adam" tests/test_optimization.py
```

And you can combine the two patterns in one:

```
pytest -k "ada and not adam" tests/test_optimization.py
```

For example to run both `test_adafactor` and `test_adam_w` you can use:

```
pytest -k "test_adam_w or test_adam_w" tests/test_optimization.py
```

Note that we use `or` here, since we want either of the keywords to match to include both.

If you want to include only tests that include both patterns, `and` is to be used:

```
pytest -k "test and ada" tests/test_optimization.py
```

Run only modified tests

You can run the tests related to the unstaged files or the current branch (according to Git) by using pytest-picked. This is a great way of quickly testing your changes didn't break anything, since it won't run the tests related to files you didn't touch.

```
pip install pytest-picked
```

```
pytest --picked
```

All tests will be run from files and folders which are modified, but not yet committed.

Automatically rerun failed tests on source modification

pytest-xdist provides a very useful feature of detecting all failed tests, and then waiting for you to modify files and continuously re-rerun those failing tests until they pass while you fix them. So that you don't need to re start pytest after you made the fix. This is repeated until all tests pass after which again a full run is performed.

```
pip install pytest-xdist
```

To enter the mode: `pytest -f` or `pytest --looponfail`

File changes are detected by looking at `looponfailroots` root directories and all of their contents (recursively). If the default for this value does not work for you, you can change it in your project by setting a configuration option in `setup.cfg`:

```
[tool:pytest]
looponfailroots = transformers tests
```

or `pytest.ini`/`tox.ini` files:

```
[pytest]
looponfailroots = transformers tests
```

This would lead to only looking for file changes in the respective directories, specified relatively to the ini-file's directory.

pytest-watch is an alternative implementation of this functionality.

Skip a test module

If you want to run all test modules, except a few you can exclude them by giving an explicit list of tests to run. For example, to run all except `test_modeling_*.py` tests:

```
pytest $(ls -1 tests/*py | grep -v test_modeling)
```

Clearing state

CI builds and when isolation is important (against speed), cache should be cleared:

```
pytest --cache-clear tests
```

Running tests in parallel

As mentioned earlier `make test` runs tests in parallel via `pytest-xdist` plugin (`-n X` argument, e.g. `-n 2` to run 2 parallel jobs).

`pytest-xdist`'s `--dist=` option allows one to control how the tests are grouped. `--dist=loadfile` puts the tests located in one file onto the same process.

Since the order of executed tests is different and unpredictable, if running the test suite with `pytest-xdist` produces failures (meaning we have some undetected coupled tests), use pytest-replay to replay the tests in the same order, which should help with then somehow reducing that failing sequence to a minimum.

Test order and repetition

It's good to repeat the tests several times, in sequence, randomly, or in sets, to detect any potential inter-dependency and state-related bugs (tear down). And the straightforward multiple repetition is just good to detect some problems that get uncovered by randomness of DL.

Repeat tests

- pytest-flakefinder:

```
pip install pytest-flakefinder
```

And then run every test multiple times (50 by default):

```
pytest --flake-finder --flake-runs=5 tests/test_failing_test.py
```

footnote: This plugin doesn't work with `-n` flag from `pytest-xdist`.

footnote: There is another plugin `pytest-repeat`, but it doesn't work with `unittest`.

Run tests in a random order

```
pip install pytest-random-order
```

Important: the presence of `pytest-random-order` will automatically randomize tests, no configuration change or command line options is required.

As explained earlier this allows detection of coupled tests - where one test's state affects the state of another. When `pytest-random-order` is installed it will print the random seed it used for that session, e.g:

```
pytest tests
[...]
Using --random-order-bucket=module
Using --random-order-seed=573663
```

So that if the given particular sequence fails, you can reproduce it by adding that exact seed, e.g.:

```
pytest --random-order-seed=573663
[...]
Using --random-order-bucket=module
Using --random-order-seed=573663
```

It will only reproduce the exact order if you use the exact same list of tests (or no list at all). Once you start to manually narrowing down the list you can no longer rely on the seed, but have to list them manually in the exact order they failed and tell pytest to not randomize them instead using `--random-order-bucket=none`, e.g.:

```
pytest --random-order-bucket=none tests/test_a.py tests/test_c.py tests/test_b.py
```

To disable the shuffling for all tests:

```
pytest --random-order-bucket=none
```

By default `--random-order-bucket=module` is implied, which will shuffle the files on the module levels. It can also shuffle on `class`, `package`, `global` and `none` levels. For the complete details please see its documentation.

Another randomization alternative is: `pytest-randomly`. This module has a very similar functionality/interface, but it doesn't have the bucket modes available in `pytest-random-order`. It has the same problem of imposing itself once installed.

Look and feel variations

pytest-sugar

pytest-sugar is a plugin that improves the look-n-feel, adds a progressbar, and show tests that fail and the assert instantly. It gets activated automatically upon installation.

```
pip install pytest-sugar
```

To run tests without it, run:

```
pytest -p no:sugar
```

or uninstall it.

Report each sub-test name and its progress

For a single or a group of tests via `pytest` (after `pip install pytest-pspec`):

```
pytest --pspec tests/test_optimization.py
```

Instantly shows failed tests

pytest-instafail shows failures and errors instantly instead of waiting until the end of test session.

```
pip install pytest-instafail
```

```
pytest --instafail
```

To GPU or not to GPU

On a GPU-enabled setup, to test in CPU-only mode add `CUDA_VISIBLE_DEVICES=""`:

```
CUDA_VISIBLE_DEVICES="" pytest tests/utils/test_logging.py
```

or if you have multiple gpus, you can specify which one is to be used by `pytest`. For example, to use only the second gpu if you have gpus `0` and `1`, you can run:

```
CUDA_VISIBLE_DEVICES="1" pytest tests/utils/test_logging.py
```

This is handy when you want to run different tasks on different GPUs.

Some tests must be run on CPU-only, others on either CPU or GPU or TPU, yet others on multiple-GPUs. The following skip decorators are used to set the requirements of tests CPU/GPU/TPU-wise:

- `require_torch` - this test will run only under torch
- `require_torch_gpu` - as `require_torch` plus requires at least 1 GPU
- `require_torch_multi_gpu` - as `require_torch` plus requires at least 2 GPUs

- `require_torch_non_multi_gpu` - as `require_torch` plus requires 0 or 1 GPUs
- `require_torch_up_to_2_gpus` - as `require_torch` plus requires 0 or 1 or 2 GPUs
- `require_torch_tpu` - as `require_torch` plus requires at least 1 TPU

Let's depict the GPU requirements in the following table:

n gpus	decorator
`>= 0`	`@require_torch`
`>= 1`	`@require_torch_gpu`
`>= 2`	`@require_torch_multi_gpu`
`< 2`	`@require_torch_non_multi_gpu`
`< 3`	`@require_torch_up_to_2_gpus`

For example, here is a test that must be run only when there are 2 or more GPUs available and pytorch is installed:

“‘python no-style from testing_utils import require_torch_multi_gpu

(**require_torch_multi_gpu?**) def test_example_with_multi_gpu():

```
These decorators can be stacked:

```python no-style
from testing_utils import require_torch_gpu

@require_torch_gpu
@some_other_decorator
def test_example_slow_on_gpu():
```

Some decorators like `@parametrized` rewrite test names, therefore `@require_*` skip decorators have to be listed last for them to work correctly. Here is an example of the correct usage:

“‘python no-style from testing_utils import require_torch_multi_gpu from parameterized import parameterized

(**parameterized.expand?**)(…) (**require_torch_multi_gpu?**) def test_integration_foo():

```
This order problem doesn't exist with `@pytest.mark.parametrize`, you can put it first or la

Inside tests:

- How many GPUs are available:
```
```

```python
from testing_utils import get_gpu_count

n_gpu = get_gpu_count()
```

Distributed training

`pytest` can't deal with distributed training directly. If this is attempted - the sub-processes don't do the right thing and end up thinking they are `pytest` and start running the test suite in loops. It works, however, if one spawns a normal process that then spawns off multiple workers and manages the IO pipes.

Here are some tests that use it:

- test_trainer_distributed.py
- test_deepspeed.py

To jump right into the execution point, search for the `execute_subprocess_async` call in those tests, which you will find inside testing_utils.py.

You will need at least 2 GPUs to see these tests in action:

```
CUDA_VISIBLE_DEVICES=0,1 RUN_SLOW=1 pytest -sv tests/test_trainer_distributed.py
```

(`RUN_SLOW` is a special decorator used by HF Transformers to normally skip heavy tests)

Output capture

During test execution any output sent to `stdout` and `stderr` is captured. If a test or a setup method fails, its according captured output will usually be shown along with the failure traceback.

To disable output capturing and to get the `stdout` and `stderr` normally, use `-s` or `--capture=no`:

```
pytest -s tests/utils/test_logging.py
```

To send test results to JUnit format output:

```
py.test tests --junitxml=result.xml
```

Color control

To have no color (e.g., yellow on white background is not readable):

```
pytest --color=no tests/utils/test_logging.py
```

Sending test report to online pastebin service

Creating a URL for each test failure:

```
pytest --pastebin=failed tests/utils/test_logging.py
```

This will submit test run information to a remote Paste service and provide a URL for each failure. You may select tests as usual or add for example -x if you only want to send one particular failure.

Creating a URL for a whole test session log:

```
pytest --pastebin=all tests/utils/test_logging.py
```

Writing tests

Most of the time if combining `pytest` and `unittest` in the same test suite works just fine. You can read here which features are supported when doing that , but the important thing to remember is that most `pytest` fixtures don't work. Neither parametrization, but we use the module `parameterized` that works in a similar way.

Parametrization

Often, there is a need to run the same test multiple times, but with different arguments. It could be done from within the test, but then there is no way of running that test for just one set of arguments.

```
# test_this1.py
import unittest
from parameterized import parameterized


class TestMathUnitTest(unittest.TestCase):
    @parameterized.expand(
        [
            ("negative", -1.5, -2.0),
            ("integer", 1, 1.0),
            ("large fraction", 1.6, 1),
        ]
    )
    def test_floor(self, name, input, expected):
        assert_equal(math.floor(input), expected)
```

Now, by default this test will be run 3 times, each time with the last 3 arguments of `test_floor` being assigned the corresponding arguments in the parameter list.

And you could run just the `negative` and `integer` sets of params with:

```
pytest -k "negative and integer" tests/test_mytest.py
```

or all but `negative` sub-tests, with:

```
pytest -k "not negative" tests/test_mytest.py
```

Besides using the `-k` filter that was just mentioned, you can find out the exact name of each sub-test and run any or all of them using their exact names.

```
pytest test_this1.py --collect-only -q
```

and it will list:

```
test_this1.py::TestMathUnitTest::test_floor_0_negative
test_this1.py::TestMathUnitTest::test_floor_1_integer
test_this1.py::TestMathUnitTest::test_floor_2_large_fraction
```

So now you can run just 2 specific sub-tests:

```
pytest test_this1.py::TestMathUnitTest::test_floor_0_negative  test_this1.py::TestMathUnitTe
```

The module parameterized works for both: `unittests` and `pytest` tests.

If, however, the test is not a `unittest`, you may use `pytest.mark.parametrize`.

Here is the same example, this time using `pytest`'s `parametrize` marker:

```
# test_this2.py
import pytest


@pytest.mark.parametrize(
    "name, input, expected",
    [
        ("negative", -1.5, -2.0),
        ("integer", 1, 1.0),
        ("large fraction", 1.6, 1),
    ],
)
def test_floor(name, input, expected):
    assert_equal(math.floor(input), expected)
```

Same as with `parameterized`, with `pytest.mark.parametrize` you can have a fine control over which sub-tests are run, if the `-k` filter doesn't do the job. Except, this parametrization function creates a slightly different set of names for the sub-tests. Here is what they look like:

```
pytest test_this2.py --collect-only -q
```

and it will list:

```
test_this2.py::test_floor[integer-1-1.0]
test_this2.py::test_floor[negative--1.5--2.0]
test_this2.py::test_floor[large fraction-1.6-1]
```

So now you can run just the specific test:

```
pytest test_this2.py::test_floor[negative--1.5--2.0] test_this2.py::test_floor[integer-1-1.0
```

as in the previous example.

Files and directories

In tests often we need to know where things are relative to the current test file, and it's not trivial since the test could be invoked from more than one directory or could reside in sub-directories with different depths. A helper class `testing_utils.TestCasePlus` solves this problem by sorting out all the basic paths and provides easy accessors to them:

- `pathlib` objects (all fully resolved):
  - `test_file_path` - the current test file path, i.e. `__file__`
  - `test_file_dir` - the directory containing the current test file
  - `tests_dir` - the directory of the `tests` test suite
  - `examples_dir` - the directory of the `examples` test suite
  - `repo_root_dir` - the directory of the repository
  - `src_dir` - the directory of `src` (i.e. where the `transformers` sub-dir resides)
- stringified paths – same as above but these return paths as strings, rather than `pathlib` objects:
  - `test_file_path_str`
  - `test_file_dir_str`
  - `tests_dir_str`
  - `examples_dir_str`
  - `repo_root_dir_str`
  - `src_dir_str`

To start using those all you need is to make sure that the test resides in a subclass of `testing_utils.TestCasePlus`. For example:

```
from testing_utils import TestCasePlus


class PathExampleTest(TestCasePlus):
    def test_something_involving_local_locations(self):
        data_dir = self.tests_dir / "fixtures/tests_samples/wmt_en_ro"
```

If you don't need to manipulate paths via `pathlib` or you just need a path as a

string, you can always invoked `str()` on the `pathlib` object or use the accessors ending with `_str`. For example:

```
from testing_utils import TestCasePlus


class PathExampleTest(TestCasePlus):
    def test_something_involving_stringified_locations(self):
        examples_dir = self.examples_dir_str
```

Temporary files and directories

Using unique temporary files and directories are essential for parallel test running, so that the tests won't overwrite each other's data. Also we want to get the temporary files and directories removed at the end of each test that created them. Therefore, using packages like `tempfile`, which address these needs is essential.

However, when debugging tests, you need to be able to see what goes into the temporary file or directory and you want to know it's exact path and not having it randomized on every test re-run.

A helper class `testing_utils.TestCasePlus` is best used for such purposes. It's a sub-class of `unittest.TestCase`, so we can easily inherit from it in the test modules.

Here is an example of its usage:

```
from testing_utils import TestCasePlus


class ExamplesTests(TestCasePlus):
    def test_whatever(self):
        tmp_dir = self.get_auto_remove_tmp_dir()
```

This code creates a unique temporary directory, and sets `tmp_dir` to its location.

- Create a unique temporary dir:

```
def test_whatever(self):
    tmp_dir = self.get_auto_remove_tmp_dir()
```

`tmp_dir` will contain the path to the created temporary dir. It will be automatically removed at the end of the test.

- Create a temporary dir of my choice, ensure it's empty before the test starts and don't empty it after the test.

```
def test_whatever(self):
    tmp_dir = self.get_auto_remove_tmp_dir("./xxx")
```

This is useful for debug when you want to monitor a specific directory and want to make sure the previous tests didn't leave any data in there.

- You can override the default behavior by directly overriding the `before` and `after` args, leading to one of the following behaviors:
  - `before=True`: the temporary dir will always be cleared at the beginning of the test.
  - `before=False`: if the temporary dir already existed, any existing files will remain there.
  - `after=True`: the temporary dir will always be deleted at the end of the test.
  - `after=False`: the temporary dir will always be left intact at the end of the test.

footnote: In order to run the equivalent of `rm -r` safely, only subdirs of the project repository checkout are allowed if an explicit `tmp_dir` is used, so that by mistake no `/tmp` or similar important part of the filesystem will get nuked. i.e. please always pass paths that start with `./`.

footnote: Each test can register multiple temporary directories and they all will get auto-removed, unless requested otherwise.

Temporary sys.path override

If you need to temporary override `sys.path` to import from another test for example, you can use the `ExtendSysPath` context manager. Example:

```
import os
from testing_utils import ExtendSysPath

bindir = os.path.abspath(os.path.dirname(__file__))
with ExtendSysPath(f"{bindir}/.."):
    from test_trainer import TrainerIntegrationCommon  # noqa
```

Skipping tests

This is useful when a bug is found and a new test is written, yet the bug is not fixed yet. In order to be able to commit it to the main repository we need make sure it's skipped during `make test`.

Methods:

- A **skip** means that you expect your test to pass only if some conditions are met, otherwise pytest should skip running the test altogether. Common examples are skipping windows-only tests on non-windows platforms, or skipping tests that depend on an external resource which is not available at the moment (for example a database).

- A **xfail** means that you expect a test to fail for some reason. A common example is a test for a feature not yet implemented, or a bug not yet fixed. When a test passes despite being expected to fail (marked with `pytest.mark.xfail`), it's an xpass and will be reported in the test summary.

One of the important differences between the two is that `skip` doesn't run the test, and `xfail` does. So if the code that's buggy causes some bad state that will affect other tests, do not use `xfail`.

Implementation

- Here is how to skip whole test unconditionally:

`python no-style @unittest.skip("this bug needs to be fixed") def test_feature_x():`

or via pytest:

`python no-style @pytest.mark.skip(reason="this bug needs to be fixed")`

or the `xfail` way:

`python no-style @pytest.mark.xfail def test_feature_x():`

Here's how to skip a test based on internal checks within the test:

```
def test_feature_x():
    if not has_something():
        pytest.skip("unsupported configuration")
```

or the whole module:

```
import pytest

if not pytest.config.getoption("--custom-flag"):
    pytest.skip("--custom-flag is missing, skipping tests", allow_module_level=True)
```

or the `xfail` way:

```
def test_feature_x():
    pytest.xfail("expected to fail until bug XYZ is fixed")
```

- Here is how to skip all tests in a module if some import is missing:

```
docutils = pytest.importorskip("docutils", minversion="0.3")
```

- Skip a test based on a condition:

`python no-style @pytest.mark.skipif(sys.version_info < (3,6), reason="requires python3.6 or higher") def test_feature_x():`

or:

```
python no-style @unittest.skipIf(torch_device == "cpu", "Can't do
half precision") def test_feature_x():
```

or skip the whole module:

```
python no-style @pytest.mark.skipif(sys.platform == 'win32',
reason="does not run on windows") class TestClass():     def
test_feature_x(self):
```

More details, example and ways are here.

Capturing outputs

Capturing the stdout/stderr output

In order to test functions that write to `stdout` and/or `stderr`, the test can access those streams using the `pytest`'s capsys system. Here is how this is accomplished:

```
import sys


def print_to_stdout(s):
    print(s)


def print_to_stderr(s):
    sys.stderr.write(s)


def test_result_and_stdout(capsys):
    msg = "Hello"
    print_to_stdout(msg)
    print_to_stderr(msg)
    out, err = capsys.readouterr()  # consume the captured output streams
    # optional: if you want to replay the consumed streams:
    sys.stdout.write(out)
    sys.stderr.write(err)
    # test:
    assert msg in out
    assert msg in err
```

And, of course, most of the time, `stderr` will come as a part of an exception, so try/except has to be used in such a case:

```
def raise_exception(msg):
    raise ValueError(msg)
```

```
def test_something_exception():
    msg = "Not a good value"
    error = ""
    try:
        raise_exception(msg)
    except Exception as e:
        error = str(e)
        assert msg in error, f"{msg} is in the exception:\n{error}"
```

Another approach to capturing stdout is via `contextlib.redirect_stdout`:

```
from io import StringIO
from contextlib import redirect_stdout


def print_to_stdout(s):
    print(s)


def test_result_and_stdout():
    msg = "Hello"
    buffer = StringIO()
    with redirect_stdout(buffer):
        print_to_stdout(msg)
    out = buffer.getvalue()
    # optional: if you want to replay the consumed streams:
    sys.stdout.write(out)
    # test:
    assert msg in out
```

An important potential issue with capturing stdout is that it may contain `\r` characters that in normal `print` reset everything that has been printed so far. There is no problem with `pytest`, but with `pytest -s` these characters get included in the buffer, so to be able to have the test run with and without `-s`, you have to make an extra cleanup to the captured output, using `re.sub(r'~.*\r', '', buf, 0, re.M)`.

But, then we have a helper context manager wrapper to automatically take care of it all, regardless of whether it has some `\r`'s in it or not, so it's a simple:

```
from testing_utils import CaptureStdout

with CaptureStdout() as cs:
    function_that_writes_to_stdout()
print(cs.out)
```

Here is a full test example:

```
from testing_utils import CaptureStdout

msg = "Secret message\r"
final = "Hello World"
with CaptureStdout() as cs:
    print(msg + final)
assert cs.out == final + "\n", f"captured: {cs.out}, expecting {final}"
```

If you'd like to capture `stderr` use the `CaptureStderr` class instead:

```
from testing_utils import CaptureStderr

with CaptureStderr() as cs:
    function_that_writes_to_stderr()
print(cs.err)
```

If you need to capture both streams at once, use the parent `CaptureStd` class:

```
from testing_utils import CaptureStd

with CaptureStd() as cs:
    function_that_writes_to_stdout_and_stderr()
print(cs.err, cs.out)
```

Also, to aid debugging test issues, by default these context managers automatically replay the captured streams on exit from the context.

Capturing logger stream

If you need to validate the output of a logger, you can use `CaptureLogger`:

```
from transformers import logging
from testing_utils import CaptureLogger

msg = "Testing 1, 2, 3"
logging.set_verbosity_info()
logger = logging.get_logger("transformers.models.bart.tokenization_bart")
with CaptureLogger(logger) as cl:
    logger.info(msg)
assert cl.out, msg + "\n"
```

Testing with environment variables

If you want to test the impact of environment variables for a specific test you can use a helper decorator `transformers.testing_utils.mockenv`

```
from testing_utils import mockenv


class HfArgumentParserTest(unittest.TestCase):
    @mockenv(TRANSFORMERS_VERBOSITY="error")
    def test_env_override(self):
        env_level_str = os.getenv("TRANSFORMERS_VERBOSITY", None)
```

At times an external program needs to be called, which requires setting `PYTHONPATH` in `os.environ` to include multiple local paths. A helper class `testing_utils.TestCasePlus` comes to help:

```
from testing_utils import TestCasePlus


class EnvExampleTest(TestCasePlus):
    def test_external_prog(self):
        env = self.get_env()
        # now call the external program, passing `env` to it
```

Depending on whether the test file was under the `tests` test suite or `examples` it'll correctly set up `env[PYTHONPATH]` to include one of these two directories, and also the `src` directory to ensure the testing is done against the current repo, and finally with whatever `env[PYTHONPATH]` was already set to before the test was called if anything.

This helper method creates a copy of the `os.environ` object, so the original remains intact.

Getting reproducible results

In some situations you may want to remove randomness for your tests. To get identical reproducible results set, you will need to fix the seed:

```
seed = 42

# python RNG
import random

random.seed(seed)

# pytorch RNGs
import torch

torch.manual_seed(seed)
torch.backends.cudnn.deterministic = True
if torch.cuda.is_available():
```

```
    torch.cuda.manual_seed_all(seed)

# numpy RNG
import numpy as np

np.random.seed(seed)

# tf RNG
tf.random.set_seed(seed)
```

Debugging tests

To start a debugger at the point of the warning, do this:

```
pytest tests/utils/test_logging.py -W error::UserWarning --pdb
```

A massive hack to create multiple pytest reports

Here is a massive `pytest` patching that I have done many years ago to aid with understanding CI reports better.

To activate it add to `tests/conftest.py` (or create it if you haven't already):

```
import pytest

def pytest_addoption(parser):
    from testing_utils import pytest_addoption_shared

    pytest_addoption_shared(parser)


def pytest_terminal_summary(terminalreporter):
    from testing_utils import pytest_terminal_summary_main

    make_reports = terminalreporter.config.getoption("--make-reports")
    if make_reports:
        pytest_terminal_summary_main(terminalreporter, id=make_reports)
```

and then when you run the test suite, add `--make-reports=mytests` like so:

```
pytest --make-reports=mytests tests
```

and it'll create 8 separate reports:

```
$ ls -1 reports/mytests/
durations.txt
```

```
errors.txt
failures_line.txt
failures_long.txt
failures_short.txt
stats.txt
summary_short.txt
warnings.txt
```

so now instead of having only a single output from `pytest` with everything together, you can now have each type of report saved into each own file.

This feature is most useful on CI, which makes it much easier to both introspect problems and also view and download individual reports.

Using a different value to `--make-reports=` for different groups of tests can have each group saved separately rather than clobbering each other.

All this functionality was already inside `pytest` but there was no way to extract it easily so I added the monkey-patching overrides testing_utils.py. Well, I did ask if I can contribute this as a feature to `pytest` but my proposal wasn't welcome.