See also the [[Writing-tests-in-PyTorch-1.8]] page for an introduction to the terms used here

Here are some tips for when you want to write tests using OpInfos. The basic layout is something like (without PEP8 vertical spacing and imports). Don't forget to instatiate the test!!!

```
class TestAdd(TestCase):
    @ops([OpInfo1, OpInfo2])
    def test_ops(self, device, dtype, op):
        for sample in op.sample_inputs(device, dtype):
            tensors = sample.input
            output = <some func>(*tensors)
            reference = op.ref(*tensors)
            assert_allclose(output, reference, atol=self.precision, rtol=self.precision)

instantiate_device_type_tests(TestAdd, globals())
```

How many tests will this run?

The `ops` decorator will create a set of tests for each `device` and `OpInfo`. The `OpInfo.dtype*` and `OpInfo.default_test_dtypes` will be combined with the `device` to further generate test functions, each one will be called with a `device`, `dtype` and `op`.

The test signature has `device`, `dtype` and `op`, where do the rest of the parameters come from?

In order to generate tensors (make_tensor is a convenient way to do that) you need some kind of **domain** of values ((high, low), non-negative, ...) and shapes, as well as optional parameters like `requires_grad` and `contiguous`. These all can be part of the OpInfo which has *Info* about the *Op* under test. The `sample_inputs` function is a convenient interface to encapsulate this knowledge.

How do I write a `sample_inputs` method to generate tensors to test with?

TBD, include examples

A test failed. How do I reproduce it?

You can rerun the test. In general, it is more convenient to use pytest than runtest.py to specify a single test. Note that the test `mytest` will be specialized, so for - `OpInfo.name=fancyop`, - `device=cpu` and - `dtype=float64`

you would run

```
python -m pytest path/to/test.py -k mytest_fancyop_cpu_float64
```

Isn't it bad form to have a loop of values inside the test?

Practicality trumps perfection. The PyTorch CI for each PR is run on many platform variants. So we end up spending about 30 minutes to build each of 17 different variants (that alone is about 8 hours of CI build time, but not the

subject of this answer), then about 11 full runs of the tests at around 1 1/2 to 2 hours apiece - another 15 hours of total CI time. Much of this runs in parallel but it is still a huge undertaking. So we prefer to have smaller, focused tests that fail early.