# Modular Benchmarking Components:

NOTE: These components are currently work in progress.

## Timer

This class is modeled on the `timeit.Timer` API, but with PyTorch specific facilities which make it more suitable for benchmarking kernels. These fall into two broad categories:

### Managing 'gotchas':

`Timer` will invoke `torch.cuda.synchronize()` if applicable, control the number of torch threads, add a warmup, and warn if a measurement appears suspect or downright unreliable.

### Integration and better measurement:

`Timer`, while modeled after the `timeit` analog, uses a slightly different API from `timeit.Timer`.

- The constructor accepts additional metadata and timing methods return a `Measurement` class rather than a float. This `Measurement` class is serializable and allows many examples to be grouped and interpreted. (See `Compare` for more details.)
- `Timer` implements the `blocked_autorange` function which is a mixture of `timeit.Timer.repeat` and `timeit.Timer.autorange`. This function selects and appropriate number and runs for a roughly fixed amount of time (like `autorange`), but is less wasteful than `autorange` which discards ~75% of measurements. It runs many times, similar to `repeat`, and returns a `Measurement` containing all of the run results.

## Compare

`Compare` takes a list of `Measurement`s in its constructor, and displays them as a formatted table for easier analysis. Identical measurements will be merged, which allows `Compare` to process replicate measurements. Several convenience methods are also provided to truncate displayed values based on the number of significant figures and color code measurements to highlight performance differences. Grouping and layout is based on metadata passed to `Timer`:

- `label`: This is a top level description. (e.g. `add`, or `multiply`) one table will be generated per unique label.
- `sub_label`: This is the label for a given configuration. Multiple statements may be logically equivalent differ in implementation. Assigning separate sub_labels will result in a row per sub_label. If a sublabel is not provided, `stmt` is used instead. Statistics (such as computing the fastest implementation) are use all sub_labels.
- `description`: This describes the inputs. For instance, `stmt=torch.add(x, y)` can be run over several values of `x` and `y`. Each pair should be given its own `description`, which allows them to appear in separate columns. Statistics do not mix values of different descriptions, since comparing the run time of drastically different inputs is generally not meaningful.
- `env`: An optional description of the torch environment. (e.g. `master` or `my_branch`). Like sub_labels, statistics are calculated across envs. (Since comparing a branch to master or a stable release is a common

use case.) However `Compare` will visually group rows which are run with the same `env`.

- `num_threads`: By default, `Timer` will run in single-threaded mode. If `Measurements` with different numbers of threads are given to `Compare`, they will be grouped into separate blocks of rows.

### Fuzzing

The `Fuzzer` class is designed to allow very flexible and repeatable construction of a wide variety of Tensors while automating away some of the tedium that comes with creating good benchmark inputs. The two APIs of interest are the constructor and `Fuzzer.take(self, n: int)`. At construction, a `Fuzzer` is a spec for the kind of Tensors that should be created. It takes a list of `FuzzedParameters`, a list of `FuzzedTensors`, and an integer with which to seed the Fuzzer.

The reason for distinguishing between parameters and Tensors is that the shapes and data of Tensors is often linked (e.g. shapes must be identical or broadcastable, indices must fall within a given range, etc.) As a result we must first materialize values for each parameter, and then use them to construct Tensors in a second pass. As a concrete reference, the following will create Tensors `x` and `y`, where `x` is a 2D Tensor and `y` is broadcastable to the shape of `x`:

```
fuzzer = Fuzzer(
  parameters=[
    FuzzedParameter("k0", 16, 16 * 1024, "loguniform"),
    FuzzedParameter("k1", 16, 16 * 1024, "loguniform"),
  ],
  tensors=[
    FuzzedTensor(
      name="x", size=("k0", "k1"), probability_contiguous=0.75
    ),
    FuzzedTensor(
      name="y", size=("k0", 1), probability_contiguous=0.75
    ),
  ],
  seed=0,
)
```

Calling `fuzzer.take(n)` will create a generator with `n` elements which yields randomly generated Tensors satisfying the above definition, as well as some metadata about the parameters and Tensors. Critically, calling `.take(...)` multiple times will produce generators which select the same parameters, allowing repeat measurements and different environments to conduct the same trial. `FuzzedParameter` and `FuzzedTensor` support a fairly involved set of behaviors to reflect the rich character of Tensor operations and representations. (For instance, note the `probability_contiguous` argument which signals that some fraction of the time non-contiguous Tensors should be created.) The best way to understand `Fuzzer`, however, is probably to experiment with `examples.fuzzer`.

# Examples:

`python -m examples.simple_timeit`

`python -m examples.compare`

```
python -m examples.fuzzer
```

```
python -m examples.end_to_end
```