# Tensor Creation API

This note describes how to create tensors in the PyTorch C++ API. It highlights the available factory functions, which populate new tensors according to some algorithm, and lists the options available to configure the shape, data type, device and other properties of a new tensor.

## Factory Functions

A *factory function* is a function that produces a new tensor. There are many factory functions available in PyTorch (both in Python and C++), which differ in the way they initialize a new tensor before returning it. All factory functions adhere to the following general "schema":

```
torch::<function-name>(<function-specific-options>, <sizes>, <tensor-options>)
```

Let's bisect the various parts of this "schema":

1. `<function-name>` is the name of the function you would like to invoke,
2. `<functions-specific-options>` are any required or optional parameters a particular factory function accepts,
3. `<sizes>` is an object of type `IntArrayRef` and specifies the shape of the resulting tensor,
4. `<tensor-options>` is an instance of `TensorOptions` and configures the data type, device, layout and other properties of the resulting tensor.

### Picking a Factory Function

The following factory functions are available at the time of this writing (the hyperlinks lead to the corresponding Python functions, since they often have more eloquent documentation -- the options are the same in C++):

- arange: Returns a tensor with a sequence of integers,
- empty: Returns a tensor with uninitialized values,
- eye: Returns an identity matrix,
- full: Returns a tensor filled with a single value,
- linspace: Returns a tensor with values linearly spaced in some interval,
- logspace: Returns a tensor with values logarithmically spaced in some interval,
- ones: Returns a tensor filled with all ones,
- rand: Returns a tensor filled with values drawn from a uniform distribution on `[0, 1)`.
- randint: Returns a tensor with integers randomly drawn from an interval,
- randn: Returns a tensor filled with values drawn from a unit normal distribution,
- randperm: Returns a tensor filled with a random permutation of integers in some interval,
- zeros: Returns a tensor filled with all zeros.

### Specifying a Size

Functions that do not require specific arguments by nature of how they fill the tensor can be invoked with just a size. For example, the following line creates a vector with 5 components, initially all set to 1:

```
torch::Tensor tensor = torch::ones(5);
```

What if we wanted to instead create a `3 x 5` matrix, or a `2 x 3 x 4` tensor? In general, an `IntArrayRef` -- the type of the size parameter of factory functions -- is constructed by specifying the size along each dimension in curly braces. For example, `{2, 3}` for a tensor (in this case matrix) with two rows and three columns, `{3, 4, 5}` for a three-dimensional tensor, and `{2}` for a one-dimensional tensor with two components. In the one dimensional case, you can omit the curly braces and just pass the single integer like we did above. Note that the squiggly braces are just one way of constructing an `IntArrayRef`. You can also pass an `std::vector<int64_t>` and a few other types. Either way, this means we can construct a three-dimensional tensor filled with values from a unit normal distribution by writing:

```
torch::Tensor tensor = torch::randn({3, 4, 5});
assert(tensor.sizes() == std::vector<int64_t>{3, 4, 5});
```

`tensor.sizes()` returns an `IntArrayRef` which can be compared against an `std::vector<int64_t>`, and we can see that it contains the sizes we passed to the tensor. You can also write `tensor.size(i)` to access a single dimension, which is equivalent to but preferred over `tensor.sizes()[i]`.

### Passing Function-Specific Parameters

Neither `ones` nor `randn` accept any additional parameters to change their behavior. One function which does require further configuration is `randint`, which takes an upper bound on the value for the integers it generates, as well as an optional lower bound, which defaults to zero. Here we create a `5 x 5` square matrix with integers between 0 and 10:

```
torch::Tensor tensor = torch::randint(/*high=*/10, {5, 5});
```

And here we raise the lower bound to 3:

```
torch::Tensor tensor = torch::randint(/*low=*/3, /*high=*/10, {5, 5});
```

The inline comments `/*low=*/` and `/*high=*/` are not required of course, but aid readability just like keyword arguments in Python.

> **Tip**
>
> The main take-away is that the size always follows the function specific arguments.

> **Attention!**
>
> Sometimes a function does not need a size at all. For example, the size of the tensor returned by `arange` is fully specified by its function-specific arguments -- the lower and upper bound of a range of integers. In that case the function does not take a `size` parameter.

## Configuring Properties of the Tensor

The previous section discussed function-specific arguments. Function-specific arguments can only change the values with which tensors are filled, and sometimes the size of the tensor. They never change things like the data type (e.g. `float32` or `int64`) of the tensor being created, or whether it lives in CPU or GPU memory. The specification of these properties is left to the very last argument to every factory function: a `TensorOptions` object, discussed below.

`TensorOptions` is a class that encapsulates the construction axes of a Tensor. With *construction axis* we mean a particular property of a Tensor that can be configured before its construction (and sometimes changed afterwards). These construction axes are:

- The `dtype` (previously "scalar type"), which controls the data type of the elements stored in the tensor,
- The `layout`, which is either strided (dense) or sparse,
- The `device`, which represents a compute device on which a tensor is stored (like a CPU or CUDA GPU),
- The `requires_grad` boolean to enable or disable gradient recording for a tensor,

If you are used to PyTorch in Python, these axes will sound very familiar. The allowed values for these axes at the moment are:

- For `dtype`: `kUInt8`, `kInt8`, `kInt16`, `kInt32`, `kInt64`, `kFloat32` and `kFloat64`,
- For `layout`: `kStrided` and `kSparse`,
- For `device`: Either `kCPU`, or `kCUDA` (which accepts an optional device index),
- For `requires_grad`: either `true` or `false`.

> **Tip**
>
> There exist "Rust-style" shorthands for dtypes, like `kF32` instead of `kFloat32`. See here for the full list.

An instance of `TensorOptions` stores a concrete value for each of these axes. Here is an example of creating a `TensorOptions` object that represents a 64-bit float, strided tensor that requires a gradient, and lives on CUDA device 1:

```
auto options =
  torch::TensorOptions()
    .dtype(torch::kFloat32)
    .layout(torch::kStrided)
    .device(torch::kCUDA, 1)
    .requires_grad(true);
```

Notice how we use the '"builder"-style methods of `TensorOptions` to construct the object piece by piece. If we pass this object as the last argument to a factory function, the newly created tensor will have these properties:

```
torch::Tensor tensor = torch::full({3, 4}, /*value=*/123, options);

assert(tensor.dtype() == torch::kFloat32);
assert(tensor.layout() == torch::kStrided);
assert(tensor.device().type() == torch::kCUDA); // or device().is_cuda()
assert(tensor.device().index() == 1);
assert(tensor.requires_grad());
```

Now, you may be thinking: do I really need to specify each axis for every new tensor I create? Fortunately, the answer is "no", as **every axis has a default value**. These defaults are:

- `kFloat32` for the dtype,
- `kStrided` for the layout,
- `kCPU` for the device,

- `false` for `requires_grad`.

What this means is that any axis you omit during the construction of a `TensorOptions` object will take on its default value. For example, this is our previous `TensorOptions` object, but with the `dtype` and `layout` defaulted:

```
auto options = torch::TensorOptions().device(torch::kCUDA, 1).requires_grad(true);
```

In fact, we can even omit all axes to get an entirely defaulted `TensorOptions` object:

```
auto options = torch::TensorOptions(); // or `torch::TensorOptions options;`
```

A nice consequence of this is that the `TensorOptions` object we just spoke so much about can be entirely omitted from any tensor factory call:

```
// A 32-bit float, strided, CPU tensor that does not require a gradient.
torch::Tensor tensor = torch::randn({3, 4});
torch::Tensor range = torch::arange(5, 10);
```

But the sugar gets sweeter: In the API presented here so far, you may have noticed that the initial `torch::TensorOptions()` is quite a mouthful to write. The good news is that for every construction axis (dtype, layout, device and `requires_grad`), there is one *free function* in the `torch::` namespace which you can pass a value for that axis. Each function then returns a `TensorOptions` object preconfigured with that axis, but allowing even further modification via the builder-style methods shown above. For example,

```
torch::ones(10, torch::TensorOptions().dtype(torch::kFloat32))
```

is equivalent to

```
torch::ones(10, torch::dtype(torch::kFloat32))
```

and further instead of

```
torch::ones(10, torch::TensorOptions().dtype(torch::kFloat32).layout(torch::kStrided))
```

we can just write

```
torch::ones(10, torch::dtype(torch::kFloat32).layout(torch::kStrided))
```

which saves us quite a bit of typing. What this means is that in practice, you should barely, if ever, have to write out `torch::TensorOptions`. Instead use the `torch::dtype()`, `torch::device()`, `torch::layout()` and `torch::requires_grad()` functions.

A final bit of convenience is that `TensorOptions` is implicitly constructible from individual values. This means that whenever a function has a parameter of type `TensorOptions`, like all factory functions do, we can directly pass a value like `torch::kFloat32` or `torch::kStrided` in place of the full object. Therefore, when there is only a single axis we would like to change compared to its default value, we can pass only that value. As such, what was

```
torch::ones(10, torch::TensorOptions().dtype(torch::kFloat32))
```

became

```
torch::ones(10, torch::dtype(torch::kFloat32))
```

and can finally be shortened to

```
torch::ones(10, torch::kFloat32)
```

Of course, it is not possible to modify further properties of the `TensorOptions` instance with this short syntax, but if all we needed was to change one property, this is quite practical.

In conclusion, we can now compare how `TensorOptions` defaults, together with the abbreviated API for creating `TensorOptions` using free functions, allow tensor creation in C++ with the same convenience as in Python. Compare this call in Python:

```
torch.randn(3, 4, dtype=torch.float32, device=torch.device('cuda', 1), requires_grad=True)
```

with the equivalent call in C++:

```
torch::randn({3, 4}, torch::dtype(torch::kFloat32).device(torch::kCUDA, 1).requires_grad(true))
```

Pretty close!

## Conversion

Just as we can use `TensorOptions` to configure how new tensors should be created, we can also use `TensorOptions` to convert a tensor from one set of properties to a new set of properties. Such a conversion usually creates a new tensor and does not occur in-place. For example, if we have a `source_tensor` created with

```
torch::Tensor source_tensor = torch::randn({2, 3}, torch::kInt64);
```

we can convert it from `int64` to `float32`:

```
torch::Tensor float_tensor = source_tensor.to(torch::kFloat32);
```

> **Attention!**
>
> The result of the conversion, `float_tensor`, is a new tensor pointing to new memory, unrelated to the source `source_tensor`.

We can then move it from CPU memory to GPU memory:

```
torch::Tensor gpu_tensor = float_tensor.to(torch::kCUDA);
```

If you have multiple CUDA devices available, the above code will copy the tensor to the *default* CUDA device, which you can configure with a `torch::DeviceGuard`. If no `DeviceGuard` is in place, this will be GPU 1. If you would like to specify a different GPU index, you can pass it to the `Device` constructor:

```
torch::Tensor gpu_two_tensor = float_tensor.to(torch::Device(torch::kCUDA, 1));
```

In the case of CPU to GPU copy and reverse, we can also configure the memory copy to be *asynchronous* by passing `/*non_blocking=*/false` as the last argument to `to()`:

```
torch::Tensor async_cpu_tensor = gpu_tensor.to(torch::kCPU, /*non_blocking=*/true);
```

## Conclusion

This note hopefully gave you a good understanding of how to create and convert tensors in an idiomatic fashion using the PyTorch C++ API. If you have any further questions or suggestions, please use our forum or GitHub issues to get in touch.