In this section, we look at the timeline of simple tensor. The content is extracted from a live presentation. It reflects the PyTorch callstacks as a snapshot on July 10, 2019. All the code refers to PyTorch location inside FB, but the opensource version points to similar locations.

Let's start with a simple tensor:

```
import torch
r = torch.rand(3,4)[0] + torch.rand(3 , 4)
```

output:

```
tensor([[0.3091, 0.5503, 1.0780, 0.9044],
        [0.5770, 0.5245, 0.3225, 1.4672],
        [0.1581, 1.0439, 0.3313, 0.9924]])
```

The code is equivalent to:

```
_t1 = torch.rand(3, 4)
_t2 = _t1.__getitem__(0)
del _t1
_t3 = torch.rand(3, 4)
r = _t2.__add__(_t3)
del _t2
del _t3
# only r remains at this point
```

Looking at them one by one:

```
_t1 = torch.rand(3, 4)  # <--- here
_t2 = _t1.__getitem__(0)
del _t1
_t3 = torch.rand(3, 4)
r = _t2.__add__(_t3)
del _t2
del _t3
```

The Python code for torch.rand doesn't exist. It all comes from

aten/src/ATen/native/native_functions.yaml

```
- func: scalar_tensor(Scalar s, *, ScalarType? dtype=None, Layout? layout=None,
    Device? device=None, bool? pin_memory=None) -> Tensor

- func: rand(int[] size, *, ScalarType? dtype=None, Layout? layout=None,
    Device? device=None, bool? pin_memory=None) -> Tensor

- func: rand(int[] size, *, Generator? generator, ScalarType? dtype=None,
    Layout? layout=None, Device? device=None, bool? pin_memory=None) -> Tensor

- func: rand(int[] size, *, Tensor(a!) out) -> Tensor(a!)
```

```
- func: rand(int[] size, *, Generator? generator, Tensor(a!) out) -> Tensor(a!)

- func: rand_like(Tensor self) -> Tensor

- func: rand_like(Tensor self, *, ScalarType dtype, Layout layout,
    Device device, bool pin_memory=False) -> Tensor
```

tools/autograd/templates/python_torch_functions.cpp

```
static PyMethodDef torch_functions[] = {
  {"arange", (PyCFunction)THPVariable_arange, METH_VARARGS | METH_KEYWORDS | METH_STATIC, NU
  {"as_tensor", (PyCFunction)THPVariable_as_tensor, METH_VARARGS | METH_KEYWORDS | METH_STAT
  {"dsmm", (PyCFunction)THPVariable_mm, METH_VARARGS | METH_KEYWORDS | METH_STATIC, NULL},
  {"from_numpy", (PyCFunction)THPVariable_from_numpy, METH_STATIC | METH_O, NULL},
  {"hsmm", (PyCFunction)THPVariable_hspmm, METH_VARARGS | METH_KEYWORDS | METH_STATIC, NULL}
  {"_promote_types", (PyCFunction)THPVariable__promote_types, METH_VARARGS | METH_KEYWORDS |
  {"nonzero", (PyCFunction)THPVariable_nonzero, METH_VARARGS | METH_KEYWORDS | METH_STATIC,
  {"randint", (PyCFunction)THPVariable_randint, METH_VARARGS | METH_KEYWORDS | METH_STATIC,
  {"range", (PyCFunction)THPVariable_range, METH_VARARGS | METH_KEYWORDS | METH_STATIC, NULL
  {"saddmm", (PyCFunction)THPVariable_sspaddmm, METH_VARARGS | METH_KEYWORDS | METH_STATIC,
  {"sparse_coo_tensor", (PyCFunction)THPVariable_sparse_coo_tensor, METH_VARARGS | METH_KEYW
  {"spmm", (PyCFunction)THPVariable_mm, METH_VARARGS | METH_KEYWORDS | METH_STATIC, NULL},
  {"tensor", (PyCFunction)THPVariable_tensor, METH_VARARGS | METH_KEYWORDS | METH_STATIC, NU
  {"get_device", (PyCFunction)THPVariable_get_device, METH_VARARGS | METH_KEYWORDS | METH_ST
  ${py_method_defs}
  {NULL}
};
```

gen/generate-code-outputs/generate-code-outputs/python_torch_functions.cpp

```
{"quantized_gru_cell", (PyCFunction)THPVariable_quantized_gru_cell, METH_VARARGS | METH_KEYW
  {"quantized_lstm", (PyCFunction)THPVariable_quantized_lstm, METH_VARARGS | METH_KEYWORDS |
  {"quantized_lstm_cell", (PyCFunction)THPVariable_quantized_lstm_cell, METH_VARARGS | METH_
  {"quantized_rnn_relu_cell", (PyCFunction)THPVariable_quantized_rnn_relu_cell, METH_VARARGS
  {"quantized_rnn_tanh_cell", (PyCFunction)THPVariable_quantized_rnn_tanh_cell, METH_VARARGS
  {"rand", (PyCFunction)THPVariable_rand, METH_VARARGS | METH_KEYWORDS | METH_STATIC, NULL},
  {"rand_like", (PyCFunction)THPVariable_rand_like, METH_VARARGS | METH_KEYWORDS | METH_STAT
  {"randint_like", (PyCFunction)THPVariable_randint_like, METH_VARARGS | METH_KEYWORDS | MET
  {"randn", (PyCFunction)THPVariable_randn, METH_VARARGS | METH_KEYWORDS | METH_STATIC, NULL
  {"randn_like", (PyCFunction)THPVariable_randn_like, METH_VARARGS | METH_KEYWORDS | METH_ST
  {"randperm", (PyCFunction)THPVariable_randperm, METH_VARARGS | METH_KEYWORDS | METH_STATIC
```

tools/autograd/templates/python_torch_functions.cpp

```
static PyTypeObject THPVariableFunctions = {
  PyVarObject_HEAD_INIT(NULL, 0)
  "torch._C._VariableFunctions",         /* tp_name */
  0,                                     /* tp_basicsize */
```

```
  0,                                    /* tp_itemsize */
  0,                                    /* tp_dealloc */
  0,                                    /* tp_print */
  0,                                    /* tp_getattr */
  0,                                    /* tp_setattr */
  0,                                    /* tp_reserved */
  0,                                    /* tp_repr */
  0,                                    /* tp_as_number */
  0,                                    /* tp_as_sequence */
  0,                                    /* tp_as_mapping */
  0,                                    /* tp_hash  */
  0,                                    /* tp_call */
  0,                                    /* tp_str */
  0,                                    /* tp_getattro */
  0,                                    /* tp_setattro */
  0,                                    /* tp_as_buffer */
  Py_TPFLAGS_DEFAULT,                   /* tp_flags */
  NULL,                                 /* tp_doc */
  0,                                    /* tp_traverse */
  0,                                    /* tp_clear */
  0,                                    /* tp_richcompare */
  0,                                    /* tp_weaklistoffset */
  0,                                    /* tp_iter */
  0,                                    /* tp_iternext */
  torch_functions,                      /* tp_methods */
  0,                                    /* tp_members */
  0,                                    /* tp_getset */
  0,                                    /* tp_base */
  0,                                    /* tp_dict */
  0,                                    /* tp_descr_get */
  0,                                    /* tp_descr_set */
  0,                                    /* tp_dictoffset */
  0,                                    /* tp_init */
  0,                                    /* tp_alloc */
  0                                     /* tp_new */
};
```

tools/autograd/templates/python_torch_functions.cpp

```
void initTorchFunctions(PyObject* module) {
  if (PyType_Ready(&THPVariableFunctions) < 0) {
    throw python_error();
  }
  Py_INCREF(&THPVariableFunctions);
  if (PyModule_AddObject(module, "_VariableFunctions",
      (PyObject*)&THPVariableFunctions) < 0) {
    throw python_error();
```

```
  }
}
```

torch/**init**.py

```
for name in dir(_C._VariableFunctions):
    if name.startswith('__'):
        continue
    globals()[name] = getattr(_C._VariableFunctions, name)
```

gen/generate-code-outputs/generate-code-outputs/python_torch_functions.cpp

```
{"rand", (PyCFunction)THPVariable_rand,
    METH_VARARGS | METH_KEYWORDS | METH_STATIC, NULL},
```

```
static PyObject * THPVariable_rand(PyObject* self_, PyObject* args,
    PyObject* kwargs)
{
  HANDLE_TH_ERRORS
  static PythonArgParser parser({
    "rand(IntArrayRef size, *, Generator generator, Tensor out=None, ScalarType dtype=None,
    "rand(IntArrayRef size, *, Tensor out=None, ScalarType dtype=None, Layout layout=torch.s
  }, /*traceable=*/true);

  ParsedArgs<8> parsed_args;
  auto r = parser.parse(args, kwargs, parsed_args);

  if (r.idx == 0) {
    if (r.isNone(2)) {
      auto size = r.intlist(0);
      auto generator = r.generator(1);
      auto dtype = r.scalartype(3);
      auto device = r.device(5);
      const auto options = TensorOptions()
          .dtype(dtype)
          .device(device)
          .layout(r.layout(4).layout)
          .requires_grad(r.toBool(7))
          .pinned_memory(r.toBool(6));
      return wrap(dispatch_rand(size, generator, options));
    } else {
      check_out_type_matches(r.tensor(2), r.scalartype(3), r.isNone(3),
                             r.layout(4), r.isNone(4),
                             r.device(5), r.isNone(5));
      return wrap(dispatch_rand(r.intlist(0), r.generator(1),
        r.tensor(2)).set_requires_grad(r.toBool(7)));
      ...
```

gen/generate-code-outputs/generate-code-outputs/python_torch_functions_dispatch.h

```
inline Tensor dispatch_rand(IntArrayRef size, Generator * generator,
    const TensorOptions & options) {
  maybe_initialize_cuda(options);
  AutoNoGIL no_gil;
  return torch::rand(size, generator, options);
}
```

gen/generate-code-outputs/generate-code-outputs/variable_factories.h

```
inline at::Tensor rand(at::IntArrayRef size, at::Generator * generator,
    const at::TensorOptions & options = {}) {
  torch::jit::Node* node = nullptr;
  std::shared_ptr<jit::tracer::TracingState> tracer_state;
  if (jit::tracer::isTracing()) {
    tracer_state = jit::tracer::getTracingState();
    at::Symbol op_name;
    op_name = jit::Symbol::fromQualString("aten::rand");
    node = tracer_state->graph->create(op_name, /*num_outputs=*/0);
    jit::tracer::recordSourceLocation(node);
    jit::tracer::addInputs(node, "size", size);
    jit::tracer::addInputs(node, "generator", generator);
    jit::tracer::addInputs(node, "options", options);
    tracer_state->graph->insertNode(node);

    jit::tracer::setTracingState(nullptr);
  }
  at::Tensor tensor = at::rand(size, generator,
    at::TensorOptions(options).is_variable(false));
  at::Tensor result =
    autograd::make_variable_consuming(std::move(tensor),
        /*requires_grad=*/options.requires_grad());
  if (tracer_state) {
    jit::tracer::setTracingState(std::move(tracer_state));
    jit::tracer::addOutput(node, result);
  }
  return result;
}
```

gen/aten/gen_aten-outputs/gen_aten-outputs/Functions.h

```
static inline Tensor rand(IntArrayRef size, Generator * generator,
    const TensorOptions & options) {
    globalLegacyTypeDispatch().initForBackend(options.backend());
    static auto table = globalATenDispatch().getOpTable(
        "aten::rand(int[] size, *, Generator? generator, "
        "ScalarType? dtype=None, Layout? layout=None, Device? device=None, "
        "bool? pin_memory=None) -> Tensor");
    return table->getOp<Tensor (IntArrayRef, Generator *, const TensorOptions &)
```

```
        >(options.backend(), options.is_variable())(size, generator, options);
}
```

gen/aten/gen_aten-outputs/gen_aten-outputs/TypeDefault.cpp

```
static auto& registerer = globalATenDispatch()
  .registerOp<Tensor (const Tensor &, bool)>(Backend::Undefined, "aten::_cast_Byte(Tensor se
  .registerOp<Tensor (const Tensor &, bool)>(Backend::Undefined, "aten::_cast_Char(Tensor se
  .registerOp<Tensor (const Tensor &, bool)>(Backend::Undefined, "aten::_cast_Double(Tensor
  .registerOp<Tensor (const Tensor &, bool)>(Backend::Undefined, "aten::_cast_Float(Tensor s
  .registerOp<Tensor (const Tensor &, bool)>(Backend::Undefined, "aten::_cast_Int(Tensor sel
  .registerOp<Tensor (IntArrayRef, Generator *, const TensorOptions &)>(Backend::Undefined,
        "aten::rand(int[] size, *, Generator? generator, ScalarType? dtype=None, "
        "Layout? layout=None, Device? device=None, bool? pin_memory=None) -> Tensor",
        &TypeDefault::rand)
```

```
Tensor TypeDefault::rand(IntArrayRef size, Generator * generator, const TensorOptions & opti
    const DeviceGuard device_guard(options.device());
    return at::native::rand(size, generator, options);
}
```

aten/src/ATen/native/TensorFactories.cpp

```
Tensor rand(IntArrayRef size, Generator* generator, const TensorOptions& options) {
  auto result = at::empty(size, options);
  return result.uniform_(0, 1, generator);
}
```

aten/src/ATen/native/native_functions.yaml

```
- func: empty(int[] size, *, ScalarType? dtype=None, Layout? layout=None,
    Device? device=None, bool? pin_memory=None) -> Tensor
  dispatch:
    CPU: empty_cpu
    CUDA: empty_cuda
    MkldnnCPU: empty_mkldnn
    SparseCPU: empty_sparse
    SparseCUDA: empty_sparse
```

aten/src/ATen/native/TensorFactories.cpp

```
Tensor empty_cpu(IntArrayRef size, const TensorOptions& options) {
  AT_ASSERT(options.backend() == Backend::CPU);
  AT_ASSERT(!options.is_variable());  // is_variable should have been 'unpacked'  // TODO: r
  check_size_nonnegative(size);

  c10::Allocator* allocator;
  if (options.pinned_memory()) {
    allocator = detail::getCUDAHooks().getPinnedMemoryAllocator();
  } else {
```

```
    allocator = at::getCPUAllocator();
  }

  int64_t nelements = prod_intlist(size);
  auto dtype = options.dtype();
  auto storage_impl = c10::make_intrusive<StorageImpl>(
    dtype,
    nelements,
    allocator->allocate(nelements * dtype.itemsize()),
    allocator,
    /*resizeable=*/true);

  auto tensor = detail::make_tensor<TensorImpl>(storage_impl, at::CPUTensorId());
  // Default TensorImpl has size [0]
  if (size.size() != 1 || size[0] != 0) {
    tensor.unsafeGetTensorImpl()->set_sizes_contiguous(size);
  }
  return tensor;
}
```

aten/src/ATen/Context.cpp

```
Allocator* getCPUAllocator() {
  return getTHDefaultAllocator();
}
```

aten/src/TH/THAllocator.cpp

```
at::Allocator* getTHDefaultAllocator() {
  return c10::GetCPUAllocator();
}
```

c10/core/CPUAllocator.cpp

```
at::Allocator* GetCPUAllocator() {
  return GetAllocator(DeviceType::CPU);
}
```

c10/core/Allocator.cpp

```
at::Allocator* GetAllocator(const at::DeviceType& t) {
  auto* alloc = allocator_array[static_cast<int>(t)];
  AT_ASSERTM(alloc, "Allocator for ", t, " is not set.");
  return alloc;
}s
```

c10/core/Allocator.h

```
template <DeviceType t>
struct AllocatorRegisterer {
  explicit AllocatorRegisterer(Allocator* alloc) {
```

```
    SetAllocator(t, alloc);
  }
};

#define REGISTER_ALLOCATOR(t, f)                    \
  namespace {                                       \
  static AllocatorRegisterer<t> g_allocator_d(f); \
  }
```

c10/core/CPUAllocator.cpp

```
REGISTER_ALLOCATOR(DeviceType::CPU, &g_cpu_alloc);

static DefaultCPUAllocator g_cpu_alloc;

struct C10_API DefaultCPUAllocator final : at::Allocator {
  DefaultCPUAllocator() {}
  ~DefaultCPUAllocator() override {}
  at::DataPtr allocate(size_t nbytes) const override {
    void* data = alloc_cpu(nbytes);
    if (FLAGS_caffe2_report_cpu_memory_usage && nbytes > 0) {
      getMemoryAllocationReporter().New(data, nbytes);
      return {data, data, &ReportAndDelete, at::Device(at::DeviceType::CPU)};
    }
    return {data, data, &free_cpu, at::Device(at::DeviceType::CPU)};
  }

void* alloc_cpu(size_t nbytes) {
  void* data;
#ifdef __ANDROID__
  data = memalign(gAlignment, nbytes);
#elif defined(_MSC_VER)
  data = _aligned_malloc(nbytes, gAlignment);
#else
  int err = posix_memalign(&data, gAlignment, nbytes);
#endif

  NUMAMove(data, nbytes, GetCurrentNUMANode());

  if (FLAGS_caffe2_cpu_allocator_do_zero_fill) {
    memset(data, 0, nbytes);
  } else if (FLAGS_caffe2_cpu_allocator_do_junk_fill) {
    memset_junk(data, nbytes);
  }

constexpr size_t gAlignment = 64;

void free_cpu(void* data) {
#ifdef _MSC_VER
  _aligned_free(data);
```

```
#else
  free(data);
#endif
}
```

aten/src/ATen/native/TensorFactories.cpp

```
Tensor empty_cpu(IntArrayRef size, const TensorOptions& options) {
  ......
  int64_t nelements = prod_intlist(size);
  auto dtype = options.dtype();
  auto storage_impl = c10::make_intrusive<StorageImpl>(
    dtype,
    nelements,
    allocator->allocate(nelements * dtype.itemsize()),
    allocator,
    /*resizeable=*/true);
```

c10/util/intrusive_ptr.h

```
template <
    class TTarget,
    class NullType = detail::intrusive_target_default_null_type<TTarget>,
    class... Args>
inline intrusive_ptr<TTarget, NullType> make_intrusive(Args&&... args) {
  return intrusive_ptr<TTarget, NullType>::make(std::forward<Args>(args)...);
}
```

```
template <
    class TTarget,
    class NullType = detail::intrusive_target_default_null_type<TTarget>>
class intrusive_ptr final {
 public:
  intrusive_ptr(const intrusive_ptr& rhs) : target_(rhs.target_) {
    retain_();
  }

  ~intrusive_ptr() noexcept {
    reset_();
  }

 private:
  TTarget* target_;

  void retain_() {
    size_t new_refcount = ++target_->refcount_;
  }
```

```
  void reset_() noexcept {
    if (target_ != NullType::singleton() && --target_->refcount_ == 0) {
      auto weak_count = --target_->weakcount_;
      const_cast<c10::guts::remove_const_t<TTarget>*>(target_)->release_resources();
      if (weak_count == 0) {
        delete target_;
      }
    }
```

```
struct C10_API StorageImpl final : public c10::intrusive_ptr_target {
```

```
class C10_API intrusive_ptr_target {
  mutable std::atomic<size_t> refcount_;
  mutable std::atomic<size_t> weakcount_;
```

c10/core/Allocator.h

```
class C10_API DataPtr {
 private:
  c10::detail::UniqueVoidPtr ptr_;
  Device device_;

 public:
  DataPtr() : ptr_(), device_(DeviceType::CPU) {}
  DataPtr(void* data, Device device) : ptr_(data), device_(device) {}
  DataPtr(void* data, void* ctx, DeleterFnPtr ctx_deleter, Device device)
      : ptr_(data, ctx, ctx_deleter), device_(device) {}
```

c10/util/UniqueVoidPtr.h

```
class UniqueVoidPtr {
 private:
  // Lifetime tied to ctx_
  void* data_;
  std::unique_ptr<void, DeleterFnPtr> ctx_;

 public:
  UniqueVoidPtr(void* data, void* ctx, DeleterFnPtr ctx_deleter)
      : data_(data), ctx_(ctx, ctx_deleter ? ctx_deleter : &deleteNothing) {}
```

c10/core/StorageImpl.h

```
struct C10_API StorageImpl final : public c10::intrusive_ptr_target {
 public:
  StorageImpl(caffe2::TypeMeta data_type, int64_t numel, at::DataPtr data_ptr,
      at::Allocator* allocator, bool resizable);

  private:
    caffe2::TypeMeta data_type_;
    DataPtr data_ptr_;
```

```
    int64_t numel_;
    bool resizable_;
    bool received_cuda_;
    Allocator* allocator_;
```

aten/src/ATen/native/TensorFactories.cpp

```
Tensor empty_cpu(IntArrayRef size, const TensorOptions& options) {
  ......
  auto tensor = detail::make_tensor<TensorImpl>(storage_impl, at::CPUTensorId());
```

aten/src/ATen/core/Tensor.h

```
class CAFFE2_API Tensor {
 protected:
  c10::intrusive_ptr<TensorImpl, UndefinedTensorImpl> impl_;

 public:
  int64_t dim() const {
    return impl_->dim();
  }
  int64_t storage_offset() const {
    return impl_->storage_offset();
  }

  Tensor abs() const;
  Tensor& abs_();
  Tensor add(const Tensor & other, Scalar alpha=1) const;
```

c10/core/TensorImpl.h

```
struct C10_API TensorImpl : public c10::intrusive_ptr_target {
 public:
  virtual int64_t dim() const;
  virtual int64_t storage_offset() const;

 private:
  Storage storage_;
#ifdef NAMEDTENSOR_ENABLED
  std::unique_ptr<c10::NamedTensorMetaInterface> named_tensor_meta_ = nullptr;
#endif
  c10::VariableVersion version_counter_;
  PyObject* pyobj_ = nullptr; // weak reference
  SmallVector<int64_t,5> sizes_;
  SmallVector<int64_t,5> strides_;
  int64_t storage_offset_ = 0;
  int64_t numel_ = 1;
  caffe2::TypeMeta data_type_;
  c10::optional<c10::Device> device_opt_;
```

```
  TensorTypeId type_id_;
  bool is_contiguous_ = true;
  bool is_wrapped_number_ = false;
  bool allow_tensor_metadata_change_ = true;
  bool reserved_ = false;
```

```
class CAFFE2_API Tensor {
    c10::intrusive_ptr<TensorImpl, UndefinedTensorImpl> impl_;
```

```
struct C10_API TensorImpl : public c10::intrusive_ptr_target {
  Storage storage_;
```

```
struct C10_API Storage {
 protected:
  c10::intrusive_ptr<StorageImpl> storage_impl_;
```

```
struct C10_API StorageImpl final : public c10::intrusive_ptr_target {
  DataPtr data_ptr_;
```

```
class C10_API DataPtr {
  c10::detail::UniqueVoidPtr ptr_;
```

```
class UniqueVoidPtr {
 std::unique_ptr<void, DeleterFnPtr> ctx_;
```

aten/src/ATen/native/TensorFactories.cpp

```
Tensor rand(IntArrayRef size, Generator* generator, const TensorOptions& options) {
  auto result = at::empty(size, options);
  return result.uniform_(0, 1, generator);
}
```

aten/src/ATen/core/TensorMethods.h

```
inline Tensor & Tensor::uniform_(double from, double to, Generator * generator) {
    static auto table = globalATenDispatch().getOpTable(
        "aten::uniform_(Tensor(a!) self, float from=0, float to=1, *, "
        "Generator? generator=None) -> Tensor(a!)");
    return table->getOp<Tensor & (Tensor &, double, double, Generator *)>(
        tensorTypeIdToBackend(type_id()),
        is_variable())(*this, from, to, generator);
}
```

aten/src/ATen/native/native_functions.yaml

```
- func: uniform_(Tensor(a!) self, float from=0, float to=1, *, Generator? generator=None) ->
  variants: method
  dispatch:
    CPU: legacy::cpu::_th_uniform_
    CUDA: uniform_cuda_
```

gen/aten/gen_aten=CPUType.cpp/CPUType.cpp

```
Tensor & CPUType::uniform_(Tensor & self, double from, double to, Generator * generator) {
#ifdef NAMEDTENSOR_ENABLED
    if (self.is_named()) {
        AT_ERROR("uniform_: no named inference rule implemented.");
    }
#endif
    const OptionalDeviceGuard device_guard(device_of(self));
    return at::native::legacy::cpu::_th_uniform_(self, from, to, generator);
}
```

aten/src/ATen/Declarations.cwrap

```
name: _th_uniform_
types:
    - floating_point
backends:
    - CPU
cname: uniform
variants: function
return: self
arguments:
    - THTensor* self
    - arg: THGenerator* GeneratorExit
```

gen/aten/gen_aten-outputs/gen_aten-outputs/LegacyTHFunctionsCPU.cpp

```
Tensor & _th_uniform_(Tensor & self, double from, double to, Generator * generator) {
    auto dispatch_scalar_type = infer_scalar_type(self);
    switch (dispatch_scalar_type) {
        case ScalarType::Float: {
            auto self_ = checked_tensor_unwrap(self,"self",1, false, Backend::CPU, ScalarTyp
            THFloatTensor_uniform(self_, generator, from, to);
            return self;
            break;
        }
```

aten/src/TH/generic/THTensorRandom.cpp

```
void THTensor_(uniform)(THTensor *self, at::Generator *_generator, double a, double b)
{
  auto gen = at::get_generator_or_default<at::CPUGenerator>(_generator, at::detail::getDefau
  at::uniform_real_distribution<float> uniform((float)a, (float)b);
  TH_TENSOR_APPLY(scalar_t, self, *self_data = (scalar_t)uniform(gen););
```

aten/src/ATen/native/TensorFactories.cpp

```
Tensor rand(IntArrayRef size, Generator* generator, const TensorOptions& options) {
  auto result = at::empty(size, options);
  return result.uniform_(0, 1, generator);
```

Move to slicing:

```
_t1 = torch.rand(3, 4)
_t2 = _t1.__getitem__(0)    # <--- here
del _t1
_t3 = torch.rand(3, 4)
r = _t2.__add__(_t3)
del _t2
del _t3
```

torch/tensor.py

```
class Tensor(torch._C._TensorBase):
```

torch/csrc/autograd/python_variable.cpp

```
PyTypeObject THPVariableType = {
  PyVarObject_HEAD_INIT(nullptr, 0)
  "torch._C._TensorBase",               /* tp_name */
  sizeof(THPVariable),                  /* tp_basicsize */
  (destructor)THPVariable_dealloc,      /* tp_dealloc */
  &THPVariable_as_mapping,              /* tp_as_mapping */
  Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE | Py_TPFLAGS_HAVE_GC, /* tp_flags */
  (traverseproc)THPVariable_traverse,   /* tp_traverse */
  (inquiry)THPVariable_clear,           /* tp_clear */
  THPVariable_properties,               /* tp_getset */
  THPVariable_pynew                     /* tp_new */
};

static PyMappingMethods THPVariable_as_mapping = {
  THPVariable_length,
  THPVariable_getitem,
  THPVariable_setitem,
};

bool THPVariable_initModule(PyObject *module)
{
  PyModule_AddObject(module, "_TensorBase",   (PyObject *)&THPVariableType);
```

torch/csrc/autograd/python_variable_indexing.cpp

```
PyObject* THPVariable_getitem(PyObject* self, PyObject* index) {
  if (index == Py_None) {
    return wrap(self_.unsqueeze(0));
  } else if (index == Py_Ellipsis) {
    return wrap(at::alias(self_));
  } else if (THPUtils_checkLong(index)) {
    return wrap(applySelect(self_, 0, THPUtils_unpackLong(index)));
  } else if (PySlice_Check(index)) {
    return wrap(applySlice(self_, 0, index, true));
```

```
  }

  // wrap index in a tuple if it's not already one
  THPObjectPtr holder = wrapTuple(index);

  variable_list variableIndices;
  Variable sliced = applySlicing(self_, holder.get(), variableIndices);
```

```
static Variable applySelect(const Variable& self, int64_t dim, int64_t index,
    int64_t real_dim=0) {
  int64_t size = self.size(dim);
  return self.select(dim, index);
}
```

aten/src/ATen/core/TensorMethods.h

```
inline Tensor Tensor::select(int64_t dim, int64_t index) const {
    static auto table = globalATenDispatch().getOpTable("aten::select(Tensor(a) self, int di
    return table->getOp<Tensor (const Tensor &, int64_t, int64_t)>(tensorTypeIdToBackend(typ
}
```

aten/src/ATen/native/native_functions.yaml

```
- func: select(Tensor(a) self, int dim, int index) -> Tensor(a)
  variants: function, method
  device_guard: False
  named_guard: False
```

gen/aten/gen_aten-outputs/gen_aten-outputs/TypeDefault.cpp

```
.registerOp<Tensor (const Tensor &, int64_t, int64_t)>(Backend::Undefined,
    "aten::select(Tensor(a) self, int dim, int index) -> Tensor(a)",
    &TypeDefault::select)
```

```
Tensor TypeDefault::select(const Tensor & self, int64_t dim, int64_t index) {
    return at::native::select(self, dim, index);
}
```

aten/src/ATen/native/TensorShape.cpp

```
Tensor select(const Tensor& self, int64_t dim, int64_t index) {
  auto sizes = self.sizes().vec();
  auto strides = self.strides().vec();
  auto storage_offset = self.storage_offset() + index * strides[dim];
  sizes.erase(sizes.begin() + dim);
  strides.erase(strides.begin() + dim);
  auto result = self.as_strided(sizes, strides, storage_offset);
```

aten/src/ATen/core/TensorMethods.h

```
inline Tensor Tensor::as_strided(IntArrayRef size, IntArrayRef stride, c10::optional<int64_t
    static auto table = globalATenDispatch().getOpTable("aten::as_strided(Tensor(a) self, in
```

```
    return table->getOp<Tensor (const Tensor &, IntArrayRef, IntArrayRef, c10::optional<int6
}
```

aten/src/ATen/native/native_functions.yaml

```
- func: as_strided(Tensor(a) self, int[] size, int[] stride, int? storage_offset=None) -> Te
  variants: function, method
  dispatch:
    CPU: as_strided_tensorimpl
    CUDA: as_strided_tensorimpl
```

aten/src/ATen/native/TensorShape.cpp

```
Tensor as_strided_tensorimpl(const Tensor& self, IntArrayRef size,
    IntArrayRef stride, optional<int64_t> storage_offset_) {
  auto storage_offset = storage_offset_.value_or(self.storage_offset());
  auto tid = self.type_id();
  auto result = detail::make_tensor<TensorImpl>(Storage(self.storage()), tid);
  setStrided(result, size, stride, storage_offset);
  return result;
}
```

c10/core/Storage.h

```
struct C10_API Storage {
 protected:
  c10::intrusive_ptr<StorageImpl> storage_impl_;
```

```
_t1 = torch.rand(3, 4)
_t2 = _t1.__getitem__(0)
del _t1                    # <--- here
_t3 = torch.rand(3, 4)
r = _t2.__add__(_t3)
del _t2
del _t3
```

torch/tensor.py

```
class Tensor(torch._C._TensorBase):
```

torch/csrc/autograd/python_variable.cpp

```
PyTypeObject THPVariableType = {
  PyVarObject_HEAD_INIT(nullptr, 0)
  "torch._C._TensorBase",                /* tp_name */
  sizeof(THPVariable),                   /* tp_basicsize */
  (destructor)THPVariable_dealloc,       /* tp_dealloc */
  &THPVariable_as_mapping,               /* tp_as_mapping */
  Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE | Py_TPFLAGS_HAVE_GC, /* tp_flags */
  (traverseproc)THPVariable_traverse,    /* tp_traverse */
  (inquiry)THPVariable_clear,            /* tp_clear */
```

```
  THPVariable_properties,                /* tp_getset */
  THPVariable_pynew                      /* tp_new */
};

static void THPVariable_dealloc(THPVariable* self)
{
  PyObject_GC_UnTrack(self);
  THPVariable_clear(self);
  self->cdata.~Variable();
  Py_TYPE(self)->tp_free((PyObject*)self);
}
```

torch/csrc/autograd/python_variable.h

```
struct THPVariable {
    PyObject_HEAD
    torch::autograd::Variable cdata;
    PyObject* backward_hooks = nullptr;
};
```

torch/csrc/autograd/variable.h

```
struct TORCH_API Variable : public at::Tensor {
```

```
class CAFFE2_API Tensor {
    c10::intrusive_ptr<TensorImpl, UndefinedTensorImpl> impl_;
```

```
struct C10_API TensorImpl : public c10::intrusive_ptr_target {
  Storage storage_;
```

```
struct C10_API Storage {
 protected:
  c10::intrusive_ptr<StorageImpl> storage_impl_;
```

```
struct C10_API StorageImpl final : public c10::intrusive_ptr_target {
  DataPtr data_ptr_;
```

```
class C10_API DataPtr {
  c10::detail::UniqueVoidPtr ptr_;
```

```
class UniqueVoidPtr {
 std::unique_ptr<void, DeleterFnPtr> ctx_;
```

```
void free_cpu(void* data) {
#ifdef _MSC_VER
  _aligned_free(data);
#else
  free(data);
#endif
}
```

The last step: addition

```
_t1 = torch.rand(3, 4)
_t2 = _t1.__getitem__(0)
del _t1
_t3 = torch.rand(3, 4)
r = _t2.__add__(_t3)       # <--- here
del _t2
del _t3
```

tools/autograd/templates/python_variable_methods.cpp

```
PyMethodDef variable_methods[] = {
  {"__add__", (PyCFunction)THPVariable_add, METH_VARARGS | METH_KEYWORDS, NULL},
  {"__radd__", (PyCFunction)THPVariable_add, METH_VARARGS | METH_KEYWORDS, NULL},
  {"__iadd__", (PyCFunction)THPVariable_add_, METH_VARARGS | METH_KEYWORDS, NULL},
```

```
bool THPVariable_initModule(PyObject *module)
{
  static std::vector<PyMethodDef> methods;
  THPUtils_addPyMethodDefs(methods, torch::autograd::variable_methods);
  PyModule_AddObject(module, "_TensorBase",   (PyObject *)&THPVariableType);
```

aten/src/ATen/native/native_functions.yaml

```
- func: add(Tensor self, Tensor other, *, Scalar alpha=1) -> Tensor
  variants: function, method
  dispatch:
    CPU: add
    CUDA: add
    SparseCPU: add
    SparseCUDA: add
    MkldnnCPU: mkldnn_add
```

gen/generate-code-outputs/generate-code-outputs/python_variable_methods.cpp

```
static PyObject * THPVariable_add(PyObject* self_, PyObject* args, PyObject* kwargs)
{
  static PythonArgParser parser({
    "add(Scalar alpha, Tensor other)|deprecated",
    "add(Tensor other, *, Scalar alpha=1)",
  });
  ParsedArgs<3> parsed_args;
  auto r = parser.parse(args, kwargs, parsed_args);

  if (r.idx == 0) {
    return wrap(dispatch_add(self, r.scalar(0), r.tensor(1)));
  } else if (r.idx == 1) {
    return wrap(dispatch_add(self, r.tensor(0), r.scalar(1)));
  }
}
```

gen/generate-code=python_torch_functions_dispatch.h/python_torch_functions_dispatch.h

```
inline Tensor dispatch_add(const Tensor & self, const Tensor & other, Scalar alpha) {
  return self.add(other, alpha);
}
```

aten/src/ATen/core/TensorMethods.h

```
inline Tensor Tensor::add(const Tensor & other, Scalar alpha) const {
    static auto table = globalATenDispatch().getOpTable(
        "aten::add(Tensor self, Tensor other, *, Scalar alpha=1) -> Tensor");
    return table->getOp<Tensor (const Tensor &, const Tensor &, Scalar)>(
        tensorTypeIdToBackend(type_id()), is_variable())(*this, other, alpha);
}
```

aten/src/ATen/native/BinaryOps.cpp

```
namespace at {
namespace native {
Tensor add(const Tensor& self, const Tensor& other, Scalar alpha) {
 Tensor result;
 auto iter = TensorIterator::binary_op(result, self, other);
 add_stub(iter->device_type(), *iter, alpha);
 return iter->output();
}
```

aten/src/ATen/native/TensorIterator.cpp

```
std::unique_ptr<TensorIterator> TensorIterator::binary_op(Tensor& out,
    const Tensor& a, const Tensor& b) {
  auto builder = TensorIterator::Builder();
  builder.add_output(out);
  builder.add_input(a);
  builder.add_input(b);
  return builder.build();
```

```
std::unique_ptr<TensorIterator> TensorIterator::Builder::build() {
  iter_->mark_outputs();
  iter_->compute_shape();
  iter_->compute_strides();
  iter_->reorder_dimensions();
  iter_->compute_types();
  iter_->allocate_outputs();
```

```
void TensorIterator::allocate_outputs() {
  for (int i = 0; i < num_outputs_; i++) {
    op.tensor = at::empty_strided(tensor_shape, tensor_stride, op.options());
  }
}
```

aten/src/ATen/native/BinaryOps.h

```
using binary_fn_alpha = void(*)(TensorIterator&, Scalar alpha);
DECLARE_DISPATCH(binary_fn_alpha, add_stub);
```

aten/src/ATen/native/cpu/BinaryOpsKernel.cpp

```
REGISTER_DISPATCH(add_stub, &add_kernel);

void add_kernel(TensorIterator& iter, Scalar alpha_scalar) {
  if (iter.dtype() == ScalarType::Bool) {
    cpu_kernel(iter, [=](bool a, bool b) -> bool { return a + b; });
  } else {
    AT_DISPATCH_ALL_TYPES(iter.dtype(), "add_cpu", [&]() {
      auto alpha = alpha_scalar.to<scalar_t>();
      auto alpha_vec = Vec256<scalar_t>(alpha);
      cpu_kernel_vec(iter,
        [=](scalar_t a, scalar_t b) -> scalar_t { return a + alpha * b; },
        [=](Vec256<scalar_t> a, Vec256<scalar_t> b) {
          return vec256::fmadd(b, alpha_vec, a);
        });
      });
  }
}
```