Conventional wisdom says that any group of programmers that makes nontrivial use of C++ defines a *subset* of the language which they are willing to use. This manifesto explicates a subset of C++ that maintains idioms from Python, the lingua franca of data science and machine learning, and serves as the basis for the API designs of both ATen and C10. Python provides an excellent basis to select which features of C++ to use, as it is unambiguous and widely understood.

Tensor is a pointer type

The Tensor class has pointer semantics; if you are a C++ programmer, think of it as being equivalent to `shared_ptr<TensorImpl>` (where TensorImpl is itself the struct that contains the metadata for the tensor). Within C++, defining a central data type in this way is *unusual*. However, this decision is well motivated by the idea of "writing Python in C++."

Consider: in idiomatic C++, a user must be able to distinguish between pointers, references and values. Python, however, only has one concept: values with pointer semantics.

- In C++, you write either `x.f()` or `x->f()` depending on if you have a pointer or a reference/value. In Python, you always write `x.f()`.
- In C++, `auto x = y` may or may not create a copy (which may or may not be deep/shallow) of an object, depending on if it's a pointer or value. In Python, `x = y` never creates a copy (unless the type in question is immutable, in which case the question of whether or not a copy was made or not is unobservable.)
- In C++, const-correctness demands a user distinguish between a mutable pointer to const data (`const int *`) and a const pointer to mutable data (`int * const`), and permits conversions which increase restrictions. In Python, mutability is an intrinsic property of an object, and an exception is raised (or mutating methods made simply unavailable) when an immutable object is mutated.

By saying that `Tensor` is a pointer type, we let users use our library effectively, without having to be well-versed in the vagaries of when to use pointers and references in C++. The resulting style is simple, intuitive and easy to explain: give tensors the `Tensor` type.

Const correctness requires different value types. In ordinary C++, one can practice "const correctness" by paying close attention to const qualifiers on pointers. There are two important variations on the humble pointer to integer type:

- `const int * x` (equivalently, `int const * x`) says that you have a mutable pointer to a constant integer; that is, you can reassign the pointer (`x = nullptr` ok), but not the integer the pointer points to (`*x = 3` errors).
- `int * const x` says that you have a constant pointer to a mutable integer; you cannot change the pointer (`x = nullptr` errors) but you can change

the integer the pointer points to (`*x = 3` ok)

The difficulty with only have the type `Tensor` in your lexicon is that there is only one variation you can concisely express: constant pointer to a mutable integer (`const Tensor`, a.k.a. `(int *) const`). No matter what const-qualifiers you place on the methods of `Tensor`, mutations are allowed through this type, because a user can easily cast the const-qualifier away by performing a copy-assignment on the tensor.

If you really want constant tensors, you could introduce a new type, `ConstTensor`, which internally stores a `const TensorImpl *`. We have not done this (because it involves a bit of boilerplate, and introduces another concept of Tensor that users have to deal with), but we are not fundamentally opposed to adding such a feature at some point in the future.

The majority of code should be un-templated

In C++, ad hoc polymorphism over types can be achieved by templates, which permit a sophisticated form of macro-expansion at every type the template parameter is instantiated at. In Python, such polymorphism is achieved by passing around opaque types and performing runtime tests to determine if the type is appropriate for a desired operation.

- In C++, a class that contains some data type T would be templated as `class C<T>`. In Python, the class we be named simply `class C`.
- In C++, a function which operates polymorphically over a type T would be templated as `template <typename T> f(...)`, so that the functions body would be re-instantiated for every type T. In Python, no such source code reinstantiation happens.

Despite these admonitions, templated programming still has a place in the lexicon of Pythonic C++. But a templated function (for user convenience) should *as quickly as possible* call a backing, untemplated function, operating on an untemplated struct, performing the dynamic type test required.

The implementation of `data()` in Tensor provides a good case study of this principle. Here is the implementation of this function:

```
void* data_ptr(); // untemplated backing function

template <typename T>
T* data() {
  C10_ASSERT(c10::dtype<T>() == dtype());
  return static_cast<T*>(data_ptr());
}
```

The templated `data()` exists because it is highly convenient to provide users a way to get an actual `int*` pointer, instead of a `void*` pointer they have to manually static cast. However, we can see that the way `data()` is implemented

is that it (1) tests that the user-requested type T (from the template parameter) indeed matches the actual dtype stored in this tensor, and then (2) carries out the static cast on the return type of `data_ptr()`, the untyped, untemplated backing function which gets the actual pointer in question.

In C++, this style has several other benefits:

1. It makes template errors more interpretable, as only a small amount of code may fail to typecheck, as opposed to a large implementation body.
2. It reduces code size, as only a small amount of code (trivially inlinable) is duplicated at every call-site.
3. It reduces compilation time, as implementations can be removed from headers and compiled only once in a cpp file.

Default to types, but support untyped idioms as well

Suppose you are writing a trainer, which demands a specific format for the output of a module, but is indifferent to what inputs you feed into this module. In Python, this would be very easy to write:

```
class Trainer:
  def __init__(self, model):
    self.model = model
  def forward(*args):
    x, y = self.model.forward(*args)
    return x.multinomial(1), x, y
```

Notice a few things about this trainer:

1. It is completely indifferent to what arguments the model takes. Anything is fine. Python has idioms that make this simple
2. It cares about the output format of the model; it expects it to have two outputs, and it's going to sample from the first tensor

How can we write this trainer in C++?

```
class Trainer {
  Module??? model_;
  std::tuple<Tensor, Tensor, Tensor> forward(??? args) {
    Tensor x, y;
    std::tie(x, y) = model_.forward(args);
    return {x.multinomial(1), x, y};
  }
}
```

As you can see from the question marks, there are immediate problems. What is the "type" of an untyped bag of arguments? What exactly is the type of forward on model_? How did we know that it was going to return exactly two tensors as outputs? Surely not all Modules behave like that? And what if you're going to subclass Trainer?

We could endlessly litigate this situation every time it comes up, or we can appeal to the principle of *writing Python in C++* and accept that a *dynamic type* solves our problem:

```
class Trainer {
  AnyModule model_;
  template <typename Args...>
  std::tuple<Tensor, Tensor, Tensor> forward(Args&&... args) {
    auto out = model_.forward<std::tuple<Tensor, Tensor>>(args);
    return {x.multinomial(1), std::get<0>(out), std::get<1>(out)};
  }
}
```

If you're calling forward on these models a lot, you might create a SampleableModule subclass which has a more type-safe type to take advantage of type safety. But we *should assume* that the dynamically typed escape hatch is available, and be willing to use it when it makes life easy.