# FX Technical Overview (WIP)

FX is a toolkit for pass writers to facilitate Python-to-Python transformation of `nn.Module` instances. This toolkit aims to support a subset of Python language semantics—rather than the whole Python language—to facilitate ease of implementation of transforms. Currently, this feature is under a Beta release and its API may change.

## Table of Contents



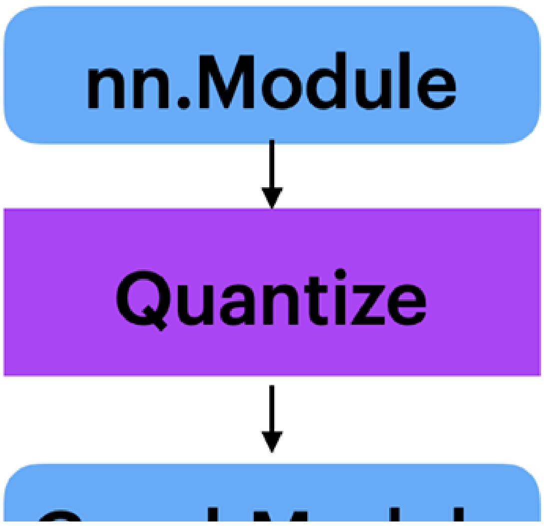
# Introduction

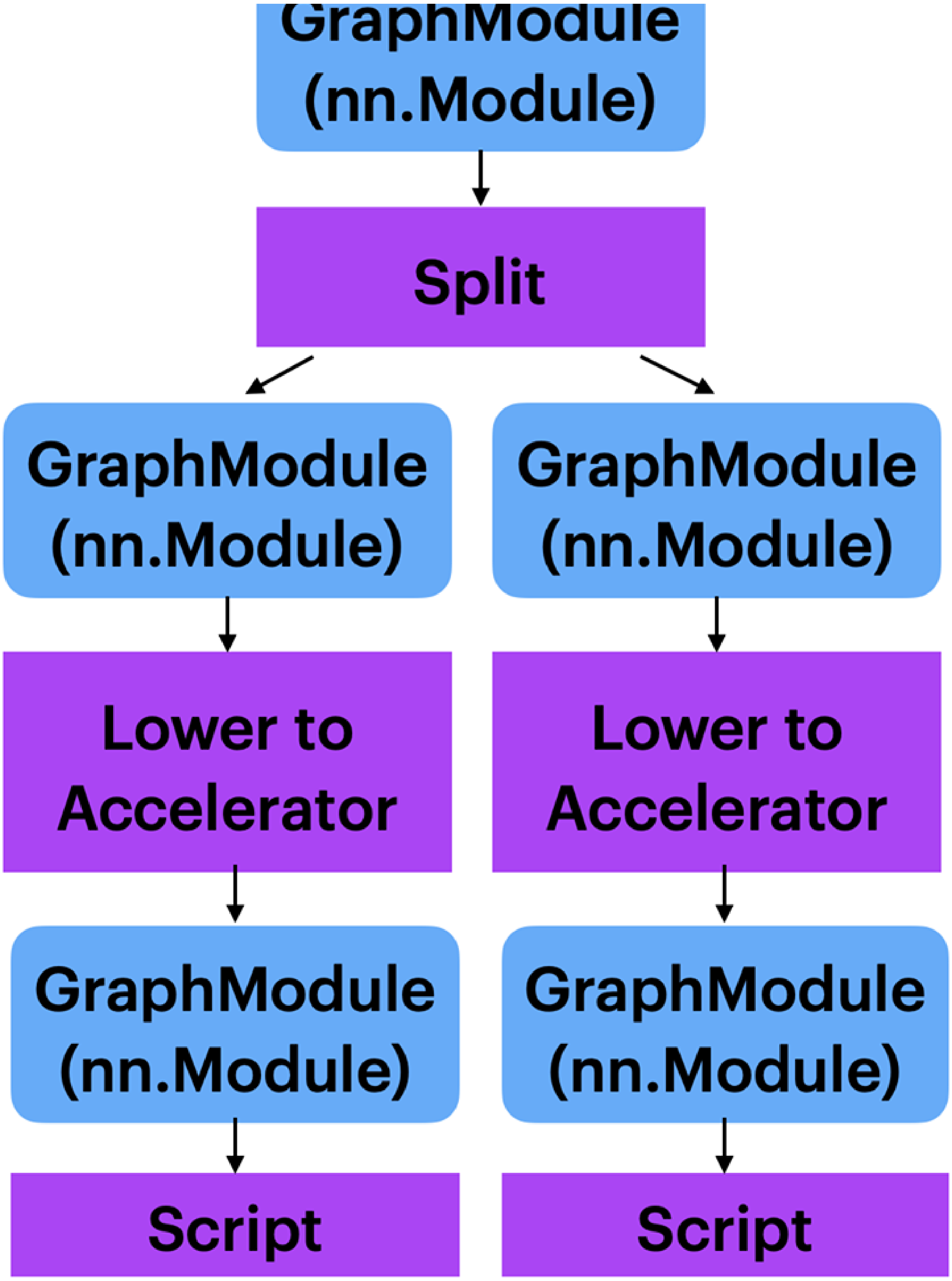
## Motivation

TODO

## Use Cases

FX should be used by pass writers to provide functionality for capturing and constructing nn.Module code in a structured way. We do not expect end users to utilize FX directly. A useful property of framing FX in this way is that passes can be seen as functions of the form `pass(in_mod : nn.Module) -> nn.Module`. This means we can create composable pipelines of transformations.






In this example pipeline, we have a Quantize transformation, which is then composed with a Split transformation, then a Lower to Accelerator transformation. Finally, the transformed Modules are compiled with TorchScript for deployment. This last point emphasizes that not only should FX transforms be composable with each other, but their products are composable with other systems like TorchScript compilation or tracing.

By using `nn.Module` as the interface between passes, FX transforms are interoperable with each other, and the resulting model can be used anywhere an `nn.Module` can be used.

### Technical Details

The following sections will walk us through the components that transform from original `torch.nn.Module` to FX IR and finally to generated Python code and a GraphModule instance:

FX's front-end makes use of the dynamic nature of Python to intercept call-sites for various entities (PyTorch operators, Module invocations, and Tensor method invocations). This functionality is exposed through an API called `torch.fx.symbolic_trace`. We can see how this works by way of an example:

```
import torch

class MyModule(torch.nn.Module):
  def __init__(self):
    super().__init__()
    self.param = torch.nn.Parameter(
        torch.rand(3, 4))
    self.linear = torch.nn.Linear(4, 5)

  def forward(self, x):
    return self.linear(x + self.param).clamp(min=0.0, max=1.0)

from torch.fx import symbolic_trace
module = MyModule()
symbolic_traced : torch.fx.GraphModule = symbolic_trace(module)

input = torch.rand(3, 4)
torch.testing.assert_allclose(symbolic_traced(input), module(input))
```

Here, we set up a simple Module that exercises different language features: fetching a parameter, applying an arithmetic operator, applying a submodule (linear), and applying a Tensor method. `symbolic_trace` returns an instance of GraphModule, which is in itself a subclass of `nn.Module`. We can see that the `symbolic_traced` instance runs and returns the same result as the original module instance module.

# Internal Structure

## Graph

TODO

## GraphModule

TODO

# Symbolic Tracing

## Tracer

`Tracer` is the class that implements the symbolic tracing functionality of `torch.fx.symbolic_trace`. A call to `symbolic_trace(m)` is equivalent to `Tracer().trace(m)`. Tracer can be subclassed to override various behaviors of the tracing process. The different behaviors that can be overridden are described in the docstrings of the methods on the class.

In the default implementation of `Tracer().trace`, the tracer first creates Proxy objects for all arguments in the `forward` function. (This happens in the call to `create_args_for_root`.) Next, the `forward` function is called with the new Proxy arguments. As the Proxies flow through the program, they record all the operations (`torch` function calls, method calls, and operators) that they touch into the growing FX Graph as Nodes.

### Proxy

Proxy objects are Node wrappers used by the Tracer to record operations seen during symbolic tracing. The mechanism through which Proxy objects record computation is `__torch_function__`. If any custom Python type defines a method named `__torch_function__`, PyTorch will invoke that `__torch_function__` implementation when an instance of that custom type is passed to a function in the `torch` namespace. In FX, when operations on Proxy are dispatched to the `__torch_function__` handler, the `__torch_function__` handler records the operation in the Graph as a Node. The Node that was recorded in the Graph is then itself wrapped in a Proxy, facilitating further application of ops on that value.

Consider the following example:

```
class M(torch.nn.Module):
    def forward(self, x):
        return torch.relu(x)

m = M()
traced = symbolic_trace(m)
```

During the call to `symbolic_trace`, the parameter `x` is transformed into a Proxy object and the corresponding Node (a Node with op = "placeholder" and target = "x") is added to the Graph. Then, the Module is run with Proxies as inputs, and recording happens via the `__torch_function__` dispatch path.

If you're doing graph transforms, you can wrap your own Proxy method around a raw Node so that you can use the overloaded operators to add additional things to a Graph.

# The FX IR

Symbolic tracing captures an intermediate representation (IR), which is represented as a doubly-linked list of Nodes.

Node is the data structure that represents individual operations within a Graph. For the most part, Nodes represent callsites to various entities, such as operators, methods, and Modules (some exceptions include Nodes that specify function inputs and outputs). Each Node has a function specified by its `op` property. The Node semantics for each value of `op` are as follows:

- `placeholder` represents a function input. The `name` attribute specifies the name this value will take on. `target` is similarly the name of the argument. `args` holds either: 1) nothing, or 2) a single argument denoting the default parameter of the function input. `kwargs` is don't-care. Placeholders correspond to the function parameters (e.g. `x`) in the graph printout.

- `get_attr` retrieves a parameter from the module hierarchy. `name` is similarly the name the result of the fetch is assigned to. `target` is the fully-qualified name of the parameter's position in the module hierarchy. `args` and `kwargs` are don't-care
- `call_function` applies a free function to some values. `name` is similarly the name of the value to assign to. `target` is the function to be applied. `args` and `kwargs` represent the arguments to the function, following the Python calling convention
- `call_module` applies a module in the module hierarchy's `forward()` method to given arguments. `name` is as previous. `target` is the fully-qualified name of the module in the module hierarchy to call. `args` and `kwargs` represent the arguments to invoke the module on, *including the self argument*.
- `call_method` calls a method on a value. `name` is as similar. `target` is the string name of the method to apply to the `self` argument. `args` and `kwargs` represent the arguments to invoke the module on, *including the self argument*
- `output` contains the output of the traced function in its `args[0]` attribute. This corresponds to the "return" statement in the Graph printout.

To facilitate easier analysis of data dependencies, Nodes have read-only properties `input_nodes` and `users`, which specify which Nodes in the Graph are used by this Node and which Nodes use this Node, respectively. Although Nodes are represented as a doubly-linked list, the use-def relationships form an acyclic graph and can be traversed as such.

# Transformation and Codegen

An invocation of `symbolic_traced` above requires a valid `forward()` method to be defined on the Module instance. How does this work? GraphModule actually generates valid Python source code based on the IR it is instantiated with. This can be seen by accessing the code attribute on the GraphModule: `print(symbolic_traced.code)`.

After symbolic tracing, the code given under Technical Details is represented as follows:

```
def forward(self, x):
    param = self.param
    add_1 = x + param;  x = param = None
    linear_1 = self.linear(add_1);  add_1 = None
    clamp_1 = linear_1.clamp(min = 0.0, max = 1.0);  linear_1 = None
    return clamp_1
```

This is the core of why FX is a Python-to-Python translation toolkit. Outside users can treat the results of FX transformations as they would any other `nn.Module` instance.