System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]jit_language_reference.rst`, line 6)

Unknown directive type "testsetup".

```
.. testsetup::

    # These are hidden from the docs, but these are necessary for `doctest`
    # since the `inspect` module doesn't play nicely with the execution
    # environment for `doctest`
    import torch

    original_script = torch.jit.script
    def script_wrapper(obj, *args, **kwargs):
        obj.__module__ = 'FakeMod'
        return original_script(obj, *args, **kwargs)

    torch.jit.script = script_wrapper

    original_trace = torch.jit.trace
    def trace_wrapper(obj, *args, **kwargs):
        obj.__module__ = 'FakeMod'
        return original_trace(obj, *args, **kwargs)

    torch.jit.trace = trace_wrapper
```

TorchScript Language Reference

TorchScript is a statically typed subset of Python that can either be written directly (using the :func:`@torch.jit.script <torch.jit.script>` decorator) or generated automatically from Python code via tracing. When using tracing, code is automatically converted into this subset of Python by recording only the actual operators on tensors and simply executing and discarding the other surrounding Python code.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]jit_language_reference.rst`, line 32); *backlink*

Unknown interpreted text role "func".

When writing TorchScript directly using `@torch.jit.script` decorator, the programmer must only use the subset of Python supported in TorchScript. This section documents what is supported in TorchScript as if it were a language reference for a stand alone language. Any features of Python not mentioned in this reference are not part of TorchScript. See *Builtin Functions* for a complete reference of available Pytorch tensor methods, modules, and functions.

As a subset of Python, any valid TorchScript function is also a valid Python function. This makes it possible to *disable TorchScript* and debug the function using standard Python tools like `pdb`. The reverse is not true: there are many valid Python programs that are not valid TorchScript programs. Instead, TorchScript focuses specifically on the features of Python that are needed to represent neural network models in PyTorch.

Types

The largest difference between TorchScript and the full Python language is that TorchScript only supports a small set of types that are needed to express neural net models. In particular, TorchScript supports:

Type	Description
`Tensor`	A PyTorch tensor of any dtype, dimension, or backend
`Tuple[T0, T1, ..., TN]`	A tuple containing subtypes `T0`, `T1`, etc. (e.g. `Tuple[Tensor, Tensor]`)
`bool`	A boolean value
`int`	A scalar integer
`float`	A scalar floating point number
`str`	A string

Type	Description
`List[T]`	A list of which all members are type `T`
`Optional[T]`	A value which is either None or type `T`
`Dict[K, V]`	A dict with key type `K` and value type `V`. Only `str`, `int`, and `float` are allowed as key types.
`T`	A TorchScript Class
`E`	A TorchScript Enum
`NamedTuple[T0, T1, ...]`	A :func:`collections.namedtuple <collections.namedtuple>` tuple type **System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]jit_language_reference.rst`, line 65); *backlink*** Unknown interpreted text role "func".
`Union[T0, T1, ...]`	One of the subtypes `T0`, `T1`, etc.

Unlike Python, each variable in TorchScript function must have a single static type. This makes it easier to optimize TorchScript functions.

Example (a type mismatch)

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]jit_language_reference.rst`, line 84)

Unknown directive type "testcode".

```
.. testcode::

    import torch

    @torch.jit.script
    def an_error(x):
        if x:
            r = torch.rand(1)
        else:
            r = 4
        return r
```

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]jit_language_reference.rst`, line 97)

Unknown directive type "testoutput".

```
.. testoutput::

     Traceback (most recent call last):
       ...
     RuntimeError: ...

     Type mismatch: r is set to type Tensor in the true branch and type int in the false branch:
     @torch.jit.script
     def an_error(x):
         if x:
         ~~~~~
             r = torch.rand(1)
             ~~~~~~~~~~~~~~~~~
         else:
         ~~~~~
             r = 4
             ~~~~~ <--- HERE
         return r
     and was used here:
         else:
             r = 4
         return r
                ~ <--- HERE...
```

Unsupported Typing Constructs

TorchScript does not support all features and types of the :mod:`typing` module. Some of these are more fundamental things that are unlikely to be added in the future while others may be added if there is enough user demand to make it a priority.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]jit_language_reference.rst`, **line 123**); ***backlink***

Unknown interpreted text role "mod".

These types and features from the :mod:`typing` module are unavailable in TorchScript.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]jit_language_reference.rst`, **line 127**); ***backlink***

Unknown interpreted text role "mod".

Item	Description
:any:`typing.Any` **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]jit_language_reference.rst`, **line 132**); ***backlink*** Unknown interpreted text role "any".	:any:`typing.Any` is currently in development but not yet released **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]jit_language_reference.rst`, **line 132**); ***backlink*** Unknown interpreted text role "any".
:any:`typing.NoReturn` **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]jit_language_reference.rst`, **line 132**); ***backlink*** Unknown interpreted text role "any".	Not implemented
:any:`typing.Sequence` **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]jit_language_reference.rst`, **line 132**); ***backlink*** Unknown interpreted text role "any".	Not implemented

Item	Description
:any:`typing.Callable` **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]jit_language_reference.rst`, **line 132**); ***backlink*** Unknown interpreted text role "any".	Not implemented
:any:`typing.Literal` **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]jit_language_reference.rst`, **line 132**); ***backlink*** Unknown interpreted text role "any".	Not implemented
:any:`typing.ClassVar` **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]jit_language_reference.rst`, **line 132**); ***backlink*** Unknown interpreted text role "any".	Not implemented
:any:`typing.Final` **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]jit_language_reference.rst`, **line 132**); ***backlink*** Unknown interpreted text role "any".	This is supported for :any:`module attributes <Module Attributes>` class attribute annotations but not for functions **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]jit_language_reference.rst`, **line 132**); ***backlink*** Unknown interpreted text role "any".
:any:`typing.AnyStr` **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]jit_language_reference.rst`, **line 132**); ***backlink*** Unknown interpreted text role "any".	TorchScript does not support :any:`bytes` so this type is not used **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]jit_language_reference.rst`, **line 132**); ***backlink*** Unknown interpreted text role "any".

Item	Description
:any:`typing.overload` **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]jit_language_reference.rst, line 132); *backlink* Unknown interpreted text role "any".	:any:`typing.overload` is currently in development but not yet released **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]jit_language_reference.rst, line 132); *backlink* Unknown interpreted text role "any".
Type aliases	Not implemented
Nominal vs structural subtyping	Nominal typing is in development, but structural typing is not
NewType	Unlikely to be implemented
Generics	Unlikely to be implemented

Any other functionality from the :any:`typing` module not explicitly listed in this documentation is unsupported.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]jit_language_reference.rst, line 146); *backlink*

Unknown interpreted text role "any".

Default Types

By default, all parameters to a TorchScript function are assumed to be Tensor. To specify that an argument to a TorchScript function is another type, it is possible to use MyPy-style type annotations using the types listed above.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]jit_language_reference.rst, line 155)

Unknown directive type "testcode".

```
.. testcode::

    import torch

    @torch.jit.script
    def foo(x, tup):
        # type: (int, Tuple[Tensor, Tensor]) -> Tensor
        t0, t1 = tup
        return t0 + t1 + x

    print(foo(3, (torch.rand(3), torch.rand(3))))
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]jit_language_reference.rst, line 167)

Unknown directive type "testoutput".

```
.. testoutput::
    :hide:

    ...
```

Note

It is also possible to annotate types with Python 3 type hints from the `typing` module.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]jit_language_reference.rst, line 176)

Unknown directive type "testcode".

```
.. testcode::

  import torch
  from typing import Tuple

  @torch.jit.script
  def foo(x: int, tup: Tuple[torch.Tensor, torch.Tensor]) -> torch.Tensor:
      t0, t1 = tup
      return t0 + t1 + x

  print(foo(3, (torch.rand(3), torch.rand(3))))
```

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]jit_language_reference.rst`, line 188)

Unknown directive type "testoutput".

```
.. testoutput::
  :hide:

  ...
```

An empty list is assumed to be `List[Tensor]` and empty dicts `Dict[str, Tensor]`. To instantiate an empty list or dict of other types, use *Python 3 type hints*.

Example (type annotations for Python 3):

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]jit_language_reference.rst`, line 200)

Unknown directive type "testcode".

```
.. testcode::

    import torch
    import torch.nn as nn
    from typing import Dict, List, Tuple

    class EmptyDataStructures(torch.nn.Module):
        def __init__(self):
            super(EmptyDataStructures, self).__init__()

        def forward(self, x: torch.Tensor) -> Tuple[List[Tuple[int, float]], Dict[str, int]]:
            # This annotates the list to be a `List[Tuple[int, float]]`
            my_list: List[Tuple[int, float]] = []
            for i in range(10):
                my_list.append((i, x.item()))

            my_dict: Dict[str, int] = {}
            return my_list, my_dict

    x = torch.jit.script(EmptyDataStructures())
```

Optional Type Refinement

TorchScript will refine the type of a variable of type `Optional[T]` when a comparison to `None` is made inside the conditional of an if-statement or checked in an `assert`. The compiler can reason about multiple `None` checks that are combined with `and`, `or`, and `not`. Refinement will also occur for else blocks of if-statements that are not explicitly written.

The `None` check must be within the if-statement's condition; assigning a `None` check to a variable and using it in the if-statement's condition will not refine the types of variables in the check. Only local variables will be refined, an attribute like `self.x` will not and must assigned to a local variable to be refined.

Example (refining types on parameters and locals):

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]jit_language_reference.rst`, line 242)

Unknown directive type "testcode".

```
.. testcode::

    import torch
    import torch.nn as nn
    from typing import Optional

    class M(nn.Module):
        z: Optional[int]

        def __init__(self, z):
            super(M, self).__init__()
            # If `z` is None, its type cannot be inferred, so it must
            # be specified (above)
            self.z = z

        def forward(self, x, y, z):
            # type: (Optional[int], Optional[int], Optional[int]) -> int
            if x is None:
                x = 1
                x = x + 1

            # Refinement for an attribute by assigning it to a local
            z = self.z
            if y is not None and z is not None:
                x = y + z

            # Refinement via an `assert`
            assert z is not None
            x += z
            return x

    module = torch.jit.script(M(2))
    module = torch.jit.script(M(None))
```

TorchScript Classes

Warning

TorchScript class support is experimental. Currently it is best suited for simple record-like types (think a `NamedTuple` with methods attached).

Python classes can be used in TorchScript if they are annotated with :func:`@torch.jit.script <torch.jit.script>`, similar to how you would declare a TorchScript function:

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]jit_language_reference.rst`, line 290); *backlink*

Unknown interpreted text role "func".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]jit_language_reference.rst`, line 293)

Unknown directive type "testcode".

```
.. testcode::
    :skipif: True  # TODO: fix the source file resolving so this can be tested

    @torch.jit.script
    class Foo:
      def __init__(self, x, y):
        self.x = x

      def aug_add_x(self, inc):
        self.x += inc
```

This subset is restricted:

- All functions must be valid TorchScript functions (including `__init__()`).

- Classes must be new-style classes, as we use `__new__()` to construct them with pybind11.
- TorchScript classes are statically typed. Members can only be declared by assigning to self in the `__init__()` method.

  For example, assigning to `self` outside of the `__init__()` method:

  ```
  @torch.jit.script
  class Foo:
    def assign_x(self):
      self.x = torch.rand(2, 3)
  ```

  Will result in:

  ```
  RuntimeError:
  Tried to set nonexistent attribute: x. Did you forget to initialize it in __init__()?:
  def assign_x(self):
    self.x = torch.rand(2, 3)
    ~~~~~~~~~~~~~~~~~~~~~~~~ <--- HERE
  ```

- No expressions except method definitions are allowed in the body of the class.
- No support for inheritance or any other polymorphism strategy, except for inheriting from `object` to specify a new-style class.

After a class is defined, it can be used in both TorchScript and Python interchangeably like any other TorchScript type:

```
# Declare a TorchScript class
@torch.jit.script
class Pair:
  def __init__(self, first, second):
    self.first = first
    self.second = second

@torch.jit.script
def sum_pair(p):
  # type: (Pair) -> Tensor
  return p.first + p.second

p = Pair(torch.rand(2, 3), torch.rand(2, 3))
print(sum_pair(p))
```

TorchScript Enums

Python enums can be used in TorchScript without any extra annotation or code:

```
from enum import Enum


class Color(Enum):
    RED = 1
    GREEN = 2

@torch.jit.script
def enum_fn(x: Color, y: Color) -> bool:
    if x == Color.RED:
        return True

    return x == y
```

After an enum is defined, it can be used in both TorchScript and Python interchangeably like any other TorchScript type. The type of the values of an enum must be `int`, `float`, or `str`. All values must be of the same type; heterogenous types for enum values are not supported.

Named Tuples

Types produced by :func:`collections.namedtuple <collections.namedtuple>` can be used in TorchScript.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]jit_language_reference.rst`, line 385); *backlink*

Unknown interpreted text role "func".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]jit_language_reference.rst`, line 387)

Unknown directive type "testcode".

```
.. testcode::
```

```
import torch
import collections

Point = collections.namedtuple('Point', ['x', 'y'])

@torch.jit.script
def total(point):
    # type: (Point) -> Tensor
    return point.x + point.y

p = Point(x=torch.rand(3), y=torch.rand(3))
print(total(p))
```

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]jit_language_reference.rst`, line 402)

Unknown directive type "testoutput".

```
.. testoutput::
    :hide:

    ...
```

Iterables

Some functions (for example, :any:`zip` and :any:`enumerate`) can only operate on iterable types. Iterable types in TorchScript include `Tensor`s, lists, tuples, dictionaries, strings, :any:`torch.nn.ModuleList` and :any:`torch.nn.ModuleDict`.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]jit_language_reference.rst`, line 413); *backlink*

Unknown interpreted text role "any".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]jit_language_reference.rst`, line 413); *backlink*

Unknown interpreted text role "any".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]jit_language_reference.rst`, line 413); *backlink*

Unknown interpreted text role "any".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]jit_language_reference.rst`, line 413); *backlink*

Unknown interpreted text role "any".

Expressions

The following Python Expressions are supported.

Literals

```
True
False
None
'string literals'
"string literals"
3  # interpreted as int
3.4  # interpreted as a float
```

List Construction

An empty list is assumed have type `List[Tensor]`. The types of other list literals are derived from the type of the members. See

Default Types for more details.

```
[3, 4]
[]
[torch.rand(3), torch.rand(4)]
```

Tuple Construction

```
(3, 4)
(3,)
```

Dict Construction

An empty dict is assumed have type `Dict[str, Tensor]`. The types of other dict literals are derived from the type of the members. See Default Types for more details.

```
{'hello': 3}
{}
{'a': torch.rand(3), 'b': torch.rand(4)}
```

Variables

See Variable Resolution for how variables are resolved.

```
my_variable_name
```

Arithmetic Operators

```
a + b
a - b
a * b
a / b
a ^ b
a @ b
```

Comparison Operators

```
a == b
a != b
a < b
a > b
a <= b
a >= b
```

Logical Operators

```
a and b
a or b
not b
```

Subscripts and Slicing

```
t[0]
t[-1]
t[0:2]
t[1:]
t[:1]
t[:]
t[0, 1]
t[0, 1:2]
t[0, :1]
t[-1, 1:, 0]
t[1:, -1, 0]
t[i:j, i]
```

Function Calls

Calls to *builtin functions*

```
torch.rand(3, dtype=torch.int)
```

Calls to other script functions:

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]jit_language_reference.rst`, line 535)

Unknown directive type "testcode".

```
.. testcode::

    import torch

    @torch.jit.script
    def foo(x):
        return x + 1

    @torch.jit.script
    def bar(x):
        return foo(x)
```

Method Calls

Calls to methods of builtin types like tensor: `x.mm(y)`

On modules, methods must be compiled before they can be called. The TorchScript compiler recursively compiles methods it sees when compiling other methods. By default, compilation starts on the `forward` method. Any methods called by `forward` will be compiled, and any methods called by those methods, and so on. To start compilation at a method other than `forward`, use the :func:`@torch.jit.export <torch.jit.export>` decorator (`forward` implicitly is marked `@torch.jit.export`).

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]jit_language_reference.rst`, line 551); *backlink*

Unknown interpreted text role "func".

Calling a submodule directly (e.g. `self.resnet(input)`) is equivalent to calling its `forward` method (e.g. `self.resnet.forward(input)`).

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]jit_language_reference.rst`, line 561)

Unknown directive type "testcode".

```
.. testcode::
    :skipif: torchvision is None

    import torch
    import torch.nn as nn
    import torchvision

    class MyModule(nn.Module):
        def __init__(self):
            super(MyModule, self).__init__()
            means = torch.tensor([103.939, 116.779, 123.68])
            self.means = torch.nn.Parameter(means.resize_(1, 3, 1, 1))
            resnet = torchvision.models.resnet18()
            self.resnet = torch.jit.trace(resnet, torch.rand(1, 3, 224, 224))

        def helper(self, input):
            return self.resnet(input - self.means)

        def forward(self, input):
            return self.helper(input)

        # Since nothing in the model calls `top_level_method`, the compiler
        # must be explicitly told to compile this method
        @torch.jit.export
        def top_level_method(self, input):
            return self.other_helper(input)

        def other_helper(self, input):
            return input + 10

    # `my_script_module` will have the compiled methods `forward`, `helper`,
    # `top_level_method`, and `other_helper`
    my_script_module = torch.jit.script(MyModule())
```

Ternary Expressions

```
x if x > y else y
```

Casts

```
float(ten)
int(3.5)
bool(ten)
str(2)``
```

Accessing Module Parameters

```
self.my_parameter
self.my_submodule.my_parameter
```

Statements

TorchScript supports the following types of statements:

Simple Assignments

```
a = b
a += b # short-hand for a = a + b, does not operate in-place on a
a -= b
```

Pattern Matching Assignments

```
a, b = tuple_or_list
a, b, *c = a_tuple
```

Multiple Assignments

```
a = b, c = tup
```

Print Statements

```
print("the result of an add:", a + b)
```

If Statements

```
if a < 4:
    r = -a
elif a < 3:
    r = a + a
else:
    r = 3 * a
```

In addition to bools, floats, ints, and Tensors can be used in a conditional and will be implicitly casted to a boolean.

While Loops

```
a = 0
while a < 4:
    print(a)
    a += 1
```

For loops with range

```
x = 0
for i in range(10):
    x *= i
```

For loops over tuples

These unroll the loop, generating a body for each member of the tuple. The body must type-check correctly for each member.

```
tup = (3, torch.rand(4))
for x in tup:
    print(x)
```

For loops over constant nn.ModuleList

To use a `nn.ModuleList` inside a compiled method, it must be marked constant by adding the name of the attribute to the `__constants__` list for the type. For loops over a `nn.ModuleList` will unroll the body of the loop at compile time, with each member of the constant module list.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]jit_language_reference.rst`, line 702)

Unknown directive type "testcode".

```
.. testcode::

    class SubModule(torch.nn.Module):
        def __init__(self):
            super(SubModule, self).__init__()
            self.weight = nn.Parameter(torch.randn(2))

        def forward(self, input):
            return self.weight + input

    class MyModule(torch.nn.Module):
        __constants__ = ['mods']

        def __init__(self):
            super(MyModule, self).__init__()
            self.mods = torch.nn.ModuleList([SubModule() for i in range(10)])

        def forward(self, v):
            for module in self.mods:
                v = module(v)
            return v


    m = torch.jit.script(MyModule())
```

Break and Continue

```
for i in range(5):
    if i == 1:
        continue
    if i == 3:
        break
    print(i)
```

Return

```
return a, b
```

Variable Resolution

TorchScript supports a subset of Python's variable resolution (i.e. scoping) rules. Local variables behave the same as in Python, except for the restriction that a variable must have the same type along all paths through a function. If a variable has a different type on different branches of an if statement, it is an error to use it after the end of the if statement.

Similarly, a variable is not allowed to be used if it is only *defined* along some paths through the function.

Example:

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]jit_language_reference.rst`, line 760)

Unknown directive type "testcode".

```
.. testcode::

    @torch.jit.script
    def foo(x):
        if x < 0:
            y = 4
        print(y)
```

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]jit_language_reference.rst`, line 768)

Unknown directive type "testoutput".

```
.. testoutput::

     Traceback (most recent call last):
       ...
     RuntimeError: ...

     y is not defined in the false branch...
     @torch.jit.script...
```

```
        def foo(x):
            if x < 0:
            ~~~~~~~~~
                y = 4
                ~~~~~ <--- HERE
            print(y)
        and was used here:
            if x < 0:
                y = 4
            print(y)
                  ~ <--- HERE...
```

Non-local variables are resolved to Python values at compile time when the function is defined. These values are then converted into TorchScript values using the rules described in Use of Python Values.

Use of Python Values

To make writing TorchScript more convenient, we allow script code to refer to Python values in the surrounding scope. For instance, any time there is a reference to `torch`, the TorchScript compiler is actually resolving it to the `torch` Python module when the function is declared. These Python values are not a first class part of TorchScript. Instead they are de-sugared at compile-time into the primitive types that TorchScript supports. This depends on the dynamic type of the Python valued referenced when compilation occurs. This section describes the rules that are used when accessing Python values in TorchScript.

Functions

TorchScript can call Python functions. This functionality is very useful when incrementally converting a model to TorchScript. The model can be moved function-by-function to TorchScript, leaving calls to Python functions in place. This way you can incrementally check the correctness of the model as you go.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]jit_language_reference.rst`, line 813)

Unknown directive type "autofunction".

```
.. autofunction:: torch.jit.is_scripting
```

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]jit_language_reference.rst`, line 815)

Unknown directive type "autofunction".

```
.. autofunction:: torch.jit.is_tracing
```

Attribute Lookup On Python Modules

TorchScript can lookup attributes on modules. *Builtin functions* like `torch.add` are accessed this way. This allows TorchScript to call functions defined in other modules.

Python-defined Constants

TorchScript also provides a way to use constants that are defined in Python. These can be used to hard-code hyper-parameters into the function, or to define universal constants. There are two ways of specifying that a Python value should be treated as a constant.

1. Values looked up as attributes of a module are assumed to be constant:

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]jit_language_reference.rst`, line 835)

Unknown directive type "testcode".

```
.. testcode::

    import math
    import torch

    @torch.jit.script
    def fn():
        return math.pi
```

2. Attributes of a ScriptModule can be marked constant by annotating them with `Final[T]`

```
import torch
import torch.nn as nn

class Foo(nn.Module):
    # `Final` from the `typing_extensions` module can also be used
    a : torch.jit.Final[int]

    def __init__(self):
        super(Foo, self).__init__()
        self.a = 1 + 4

    def forward(self, input):
        return self.a + input

f = torch.jit.script(Foo())
```

Supported constant Python types are

- `int`
- `float`
- `bool`
- `torch.device`
- `torch.layout`
- `torch.dtype`
- tuples containing supported types
- `torch.nn.ModuleList` which can be used in a TorchScript for loop

Module Attributes

The `torch.nn.Parameter` wrapper and `register_buffer` can be used to assign tensors to a module. Other values assigned to a module that is compiled will be added to the compiled module if their types can be inferred. All types available in TorchScript can be used as module attributes. Tensor attributes are semantically the same as buffers. The type of empty lists and dictionaries and `None` values cannot be inferred and must be specified via PEP 526-style class annotations. If a type cannot be inferred and is not explicitly annotated, it will not be added as an attribute to the resulting :class:`ScriptModule`.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]jit_language_reference.rst`, line 883); *backlink*

Unknown interpreted text role "class".

Example:

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]jit_language_reference.rst`, line 895)

Unknown directive type "testcode".

```
.. testcode::

    from typing import List, Dict

    class Foo(nn.Module):
        # `words` is initialized as an empty list, so its type must be specified
        words: List[str]

        # The type could potentially be inferred if `a_dict` (below) was not
        # empty, but this annotation ensures `some_dict` will be made into the
        # proper type
        some_dict: Dict[str, int]

        def __init__(self, a_dict):
            super(Foo, self).__init__()
            self.words = []
            self.some_dict = a_dict

            # `int`s can be inferred
            self.my_int = 10

        def forward(self, input):
            # type: (str) -> int
            self.words.append(input)
            return self.some_dict[input] + self.my_int
```

```
f = torch.jit.script(Foo({'hi': 2}))
```