TorchScript Language Reference

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]jit_language_reference_v2.rst`, line 1)

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
```

This reference manual describes the syntax and core semantics of the TorchScript language. TorchScript is a statically typed subset of the Python language. This document explains the supported features of Python in TorchScript and also how the language diverges from regular Python. Any features of Python that are not mentioned in this reference manual are not part of TorchScript. TorchScript focuses specifically on the features of Python that are needed to represent neural network models in PyTorch.

Terminology

This document uses the following terminologies:

Pattern	**Notes**
`::=`	Indicates that the given symbol is defined as.
`" "`	Represents real keywords and delimiters that are part of the syntax.
`A	B`	Indicates either A or B.
`( )`	Indicates grouping.
`[]`	Indicates optional.
`A+`	Indicates a regular expression where term A is repeated at least once.
`A*`	Indicates a regular expression where term A is repeated zero or more times.

Type System

TorchScript is a statically typed subset of Python. 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.

TorchScript Types

The TorchScript type system consists of `TSType` and `TSModuleType` as defined below.

```
TSAllType ::= TSType | TSModuleType
TSType    ::= TSMetaType | TSPrimitiveType | TSStructuralType | TSNominalType
```

`TSType` represents the majority of TorchScript types that are composable and that can be used in TorchScript type annotations. `TSType` refers to any of the following:

- Meta Types, e.g., `Any`
- Primitive Types, e.g., `int`, `float`, and `str`
- Structural Types, e.g., `Optional[int]` or `List[MyClass]`
- Nominal Types (Python classes), e.g., `MyClass` (user-defined), `torch.tensor` (built-in)

`TSModuleType` represents `torch.nn.Module` and its subclasses. It is treated differently from `TSType` because its type schema is inferred partly from the object instance and partly from the class definition. As such, instances of a `TSModuleType` may not follow the same static type schema. `TSModuleType` cannot be used as a TorchScript type annotation or be composed with `TSType` for type safety considerations.

Meta Types

Meta types are so abstract that they are more like type constraints than concrete types. Currently TorchScript defines one meta-type, `Any`, that represents any TorchScript type.

`Any` Type

The `Any` type represents any TorchScript type. `Any` specifies no type constraints, thus there is no type-checking on `Any`. As such it can be bound to any Python or TorchScript data types (e.g., `int`, TorchScript `tuple`, or an arbitrary Python class that is not scripted).

```
TSMetaType ::= "Any"
```

Where:

- `Any` is the Python class name from the typing module. Therefore, to use the `Any` type, you must import it from `typing` (e.g., `from typing import Any`).
- Since `Any` can represent any TorchScript type, the set of operators that are allowed to operate on values of this type on `Any` is limited.

Operators Supported for `Any` Type

- Assignment to data of `Any` type.
- Binding to parameter or return of `Any` type.
- `x is`, `x is not` where `x` is of `Any` type.
- `isinstance(x, Type)` where `x` is of `Any` type.
- Data of `Any` type is printable.
- Data of `List[Any]` type may be sortable if the data is a list of values of the same type `T` and that `T` supports comparison operators.

Compared to Python

`Any` is the least constrained type in the TorchScript type system. In that sense, it is quite similar to the `Object` class in Python. However, `Any` only supports a subset of the operators and methods that are supported by `Object`.

Design Notes

When we script a PyTorch module, we may encounter data that is not involved in the execution of the script. Nevertheless, it has to be described by a type schema. It is not only cumbersome to describe static types for unused data (in the context of the script), but also may lead to unnecessary scripting failures. `Any` is introduced to describe the type of the data where precise static types are not necessary for compilation.

Example 1

This example illustrates how `Any` can be used to allow the second element of the tuple parameter to be of any type. This is possible because `x[1]` is not involved in any computation that requires knowing its precise type.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]jit_language_reference_v2.rst`, line 140)

Unknown directive type "testcode".

```
.. testcode::

    import torch

    from typing import Tuple
    from typing import Any

    @torch.jit.export
    def inc_first_element(x: Tuple[int, Any]):
        return (x[0]+1, x[1])

    m = torch.jit.script(inc_first_element)
    print(m((1,2.0)))
    print(m((1,(100,200))))
```

The example above produces the following output:

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]jit_language_reference_v2.rst`, line 157)

Unknown directive type "testoutput".

```
.. testoutput::

    (2, 2.0)
    (2, (100, 200))
```

The second element of the tuple is of `Any` type, thus can bind to multiple types. For example, `(1, 2.0)` binds a float type to `Any` as in `Tuple[int, Any]`, whereas `(1, (100, 200))` binds a tuple to `Any` in the second invocation.

Example 2

This example illustrates how we can use `isinstance` to dynamically check the type of the data that is annotated as `Any` type:

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]jit_language_reference_v2.rst`, line 171)

Unknown directive type "testcode".

```
.. testcode::

    import torch
    from typing import Any

    def f(a:Any):
        print(a)
        return (isinstance(a, torch.Tensor))

    ones = torch.ones([2])
    m = torch.jit.script(f)
    print(m(ones))
```

The example above produces the following output:

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]jit_language_reference_v2.rst`, line 186)

Unknown directive type "testoutput".

```
.. testoutput::

     1
     1
    [ CPUFloatType{2} ]
    True
```

Primitive Types

Primitive TorchScript types are types that represent a single type of value and go with a single pre-defined type name.

```
TSPrimitiveType ::= "int" | "float" | "double" | "complex" | "bool" | "str" | "None"
```

Structural Types

Structural types are types that are structurally defined without a user-defined name (unlike nominal types), such as `Future[int]`. Structural types are composable with any `TSType`.

```
TSStructuralType ::=  TSTuple | TSNamedTuple | TSList | TSDict |
                     TSOptional | TSUnion | TSFuture | TSRRef

TSTuple          ::= "Tuple" "[" (TSType ",")* TSType "]"
TSNamedTuple     ::= "namedtuple" "(" (TSType ",")* TSType ")"
TSList           ::= "List" "[" TSType "]"
TSOptional       ::= "Optional" "[" TSType "]"
TSUnion          ::= "Union" "[" (TSType ",")* TSType "]"
TSFuture         ::= "Future" "[" TSType "]"
TSRRef           ::= "RRef" "[" TSType "]"
TSDict           ::= "Dict" "[" KeyType "," TSType "]"
KeyType          ::= "str" | "int" | "float" | "bool" | TensorType | "Any"
```

Where:

- `Tuple`, `List`, `Optional`, `Union`, `Future`, `Dict` represent Python type class names that are defined in the module `typing`. To use these type names, you must import them from `typing` (e.g., `from typing import Tuple`).
- `namedtuple` represents the Python class `collections.namedtuple` or `typing.NamedTuple`.
- `Future` and `RRef` represent the Python classes `torch.futures` and `torch.distributed.rpc`.

Compared to Python

Apart from being composable with TorchScript types, these TorchScript structural types often support a common subset of the operators and methods of their Python counterparts.

Example 1

This example uses `typing.NamedTuple` syntax to define a tuple:

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]jit_language_reference_v2.rst`, line 238)

Unknown directive type "testcode".

```
.. testcode::

    import torch
    from typing import NamedTuple
    from typing import Tuple
```

```
class MyTuple(NamedTuple):
    first: int
    second: int

def inc(x: MyTuple) -> Tuple[int, int]:
    return (x.first+1, x.second+1)

t = MyTuple(first=1, second=2)
scripted_inc = torch.jit.script(inc)
print("TorchScript:", scripted_inc(t))
```

The example above produces the following output:

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]jit_language_reference_v2.rst`, line 257)

Unknown directive type "testoutput".

```
.. testoutput::

    TorchScript: (2, 3)
```

Example 2

This example uses `collections.namedtuple` syntax to define a tuple:

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]jit_language_reference_v2.rst`, line 265)

Unknown directive type "testcode".

```
.. testcode::

    import torch
    from typing import NamedTuple
    from typing import Tuple
    from collections import namedtuple

    _AnnotatedNamedTuple = NamedTuple('_NamedTupleAnnotated', [('first', int), ('second', int)])
    _UnannotatedNamedTuple = namedtuple('_NamedTupleAnnotated', ['first', 'second'])

    def inc(x: _AnnotatedNamedTuple) -> Tuple[int, int]:
        return (x.first+1, x.second+1)

    m = torch.jit.script(inc)
    print(inc(_UnannotatedNamedTuple(1,2)))
```

The example above produces the following output:

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]jit_language_reference_v2.rst`, line 283)

Unknown directive type "testoutput".

```
.. testoutput::

    (2, 3)
```

Example 3

This example illustrates a common mistake of annotating structural types, i.e., not importing the composite type classes from the `typing` module:

```
import torch

# ERROR: Tuple not recognized because not imported from typing
@torch.jit.export
def inc(x: Tuple[int, int]):
    return (x[0]+1, x[1]+1)

m = torch.jit.script(inc)
print(m((1,2)))
```

Running the above code yields the following scripting error:

```
File "test-tuple.py", line 5, in <module>
    def inc(x: Tuple[int, int]):
NameError: name 'Tuple' is not defined
```

The remedy is to add the line `from typing import Tuple` to the beginning of the code.

Nominal Types

Nominal TorchScript types are Python classes. These types are called nominal because they are declared with a custom name and are compared using class names. Nominal classes are further classified into the following categories:

```
TSNominalType ::= TSBuiltinClasses | TSCustomClass | TSEnum
```

Among them, `TSCustomClass` and `TSEnum` must be compilable to TorchScript Intermediate Representation (IR). This is enforced by the type-checker.

Built-in Class

Built-in nominal types are Python classes whose semantics are built into the TorchScript system (e.g., tensor types). TorchScript defines the semantics of these built-in nominal types, and often supports only a subset of the methods or attributes of its Python class definition.

```
TSBuiltinClass ::= TSTensor | "torch.device" | "torch.Stream" | "torch.dtype" |
                   "torch.nn.ModuleList" | "torch.nn.ModuleDict" | ...
TSTensor       ::= "torch.Tensor" | "common.SubTensor" | "common.SubWithTorchFunction" |
                   "torch.nn.parameter.Parameter" | and subclasses of torch.Tensor
```

Special Note on torch.nn.ModuleList and torch.nn.ModuleDict

Although `torch.nn.ModuleList` and `torch.nn.ModuleDict` are defined as a list and dictionary in Python, they behave more like tuples in TorchScript:

- In TorchScript, instances of `torch.nn.ModuleList` or `torch.nn.ModuleDict` are immutable.
- Code that iterates over `torch.nn.ModuleList` or `torch.nn.ModuleDict` is completely unrolled so that elements of `torch.nn.ModuleList` or keys of `torch.nn.ModuleDict` can be of different subclasses of `torch.nn.Module`.

Example

The following example highlights the use of a few built-in Torchscript classes (`torch.*`):

```
import torch

@torch.jit.script
class A:
    def __init__(self):
        self.x = torch.rand(3)

    def f(self, y: torch.device):
        return self.x.to(device=y)

def g():
    a = A()
    return a.f(torch.device("cpu"))

script_g = torch.jit.script(g)
print(script_g.graph)
```

Custom Class

Unlike built-in classes, semantics of custom classes are user-defined and the entire class definition must be compilable to TorchScript IR and subject to TorchScript type-checking rules.

```
TSClassDef ::= [ "@torch.jit.script" ]
                 "class" ClassName [ "(object)" ]  ":"
                    MethodDefinition |
                [ "@torch.jit.ignore" ] | [ "@torch.jit.unused" ]
                    MethodDefinition
```

Where:

- Classes must be new-style classes. Python 3 supports only new-style classes. In Python 2.x, a new-style class is specified by subclassing from the object.
- Instance data attributes are statically typed, and instance attributes must be declared by assignments inside the `__init__()` method.
- Method overloading is not supported (i.e., you cannot have multiple methods with the same method name).
- `MethodDefinition` must be compilable to TorchScript IR and adhere to TorchScriptâ€™s type-checking rules, (i.e., all methods must be valid TorchScript functions and class attribute definitions must be valid TorchScript statements).
- `torch.jit.ignore` and `torch.jit.unused` can be used to ignore the method or function that is not fully torchscriptable or should be ignored by the compiler.

Compared to Python

TorchScript custom classes are quite limited compared to their Python counterpart. Torchscript custom classes:

- Do not support class attributes.
- Do not support subclassing except for subclassing an interface type or object.
- Do not support method overloading.
- Must initialize all its instance attributes in `__init__()`; this is because TorchScript constructs a static schema of the class by inferring attribute types in `__init__()`.
- Must contain only methods that satisfy TorchScript type-checking rules and are compilable to TorchScript IRs.

Example 1

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

```
@torch.jit.script
class MyClass:
    def __init__(self, x: int):
```

```
        self.x = x

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

Example 2

A TorchScript custom class type must "declare" all its instance attributes by assignments in `__init__()`. If an instance attribute is not defined in `__init__()` but accessed in other methods of the class, the class cannot be compiled as a TorchScript class, as shown in the following example:

```
import torch

@torch.jit.script
class foo:
    def __init__(self):
        self.y = 1

# ERROR: self.x is not defined in __init__
def assign_x(self):
    self.x = torch.rand(2, 3)
```

The class will fail to compile and issue the following error:

```
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
```

Example 3

In this example, a TorchScript custom class defines a class variable name, which is not allowed:

```
import torch

@torch.jit.script
class MyClass(object):
    name = "MyClass"
    def __init__(self, x: int):
        self.x = x

def fn(a: MyClass):
    return a.name
```

It leads to the following compile-time error:

```
RuntimeError:
'__torch__.MyClass' object has no attribute or method 'name'. Did you forget to initialize an attribute in __in
    File "test-class2.py", line 10
def fn(a: MyClass):
    return a.name
        ~~~~~~ <--- HERE
```

Enum Type

Like custom classes, semantics of the enum type are user-defined and the entire class definition must be compilable to TorchScript IR and adhere to TorchScript type-checking rules.

```
TSEnumDef ::= "class" Identifier "(enum.Enum | TSEnumType)" ":"
               ( MemberIdentifier "=" Value )+
               ( MethodDefinition )*
```

Where:

- Value must be a TorchScript literal of type `int`, `float`, or `str`, and must be of the same TorchScript type.
- `TSEnumType` is the name of a TorchScript enumerated type. Similar to Python enum, TorchScript allows restricted `Enum` subclassing, that is, subclassing an enumerated is allowed only if it does not define any members.

Compared to Python

- TorchScript supports only `enum.Enum`. It does not support other variations such as `enum.IntEnum`, `enum.Flag`, `enum.IntFlag`, and `enum.auto`.
- Values of TorchScript enum members must be of the same type and can only be `int`, `float`, or `str` types, whereas Python enum members can be of any type.
- Enums containing methods are ignored in TorchScript.

Example 1

The following example defines the class `Color` as an `Enum` type:

```
import torch
from enum import Enum

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

def enum_fn(x: Color, y: Color) -> bool:
    if x == Color.RED:
        return True
    return x == y
```

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

print("Eager: ", enum_fn(Color.RED, Color.GREEN))
print("TorchScript: ", m(Color.RED, Color.GREEN))
```

Example 2

The following example shows the case of restricted enum subclassing, where `BaseColor` does not define any member, thus can be subclassed by `Color`:

```
import torch
from enum import Enum

class BaseColor(Enum):
    def foo(self):
        pass

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

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

m = torch.jit.script(enum_fn)

print("TorchScript: ", m(Color.RED, Color.GREEN))
print("Eager: ", enum_fn(Color.RED, Color.GREEN))
```

TorchScript Module Class

`TSModuleType` is a special class type that is inferred from object instances that are created outside TorchScript. `TSModuleType` is named by the Python class of the object instance. The `__init__()` method of the Python class is not considered a TorchScript method, so it does not have to comply with TorchScriptâ€™s type-checking rules.

The type schema of a module instance class is constructed directly from an instance object (created outside the scope of TorchScript) rather than inferred from `__init__()` like custom classes. It is possible that two objects of the same instance class type follow two different type schemas.

In this sense, `TSModuleType` is not really a static type. Therefore, for type safety considerations, `TSModuleType` cannot be used in a TorchScript type annotation or be composed with `TSType`.

Module Instance Class

TorchScript module type represents the type schema of a user-defined PyTorch module instance. When scripting a PyTorch module, the module object is always created outside TorchScript (i.e., passed in as parameter to `forward`). The Python module class is treated as a module instance class, so the `__init__()` method of the Python module class is not subject to the type-checking rules of TorchScript.

```
TSModuleType ::= "class" Identifier "(torch.nn.Module)" ":"
                    ClassBodyDefinition
```

Where:

- `forward()` and other methods decorated with `@torch.jit.export` must be compilable to TorchScript IR and subject to TorchScriptâ€™s type-checking rules.

Unlike custom classes, only the forward method and other methods decorated with `@torch.jit.export` of the module type need to be compilable. Most notably, `__init__()` is not considered a TorchScript method. Consequently, module type constructors cannot be invoked within the scope of TorchScript. Instead, TorchScript module objects are always constructed outside and passed into `torch.jit.script(ModuleObj)`.

Example 1

This example illustrates a few features of module types:

- The `TestModule` instance is created outside the scope of TorchScript (i.e., before invoking `torch.jit.script`).
- `__init__()` is not considered a TorchScript method, therefore, it does not have to be annotated and can contain arbitrary Python code. In addition, the `__init__()` method of an instance class cannot be invoked in TorchScript code. Because `TestModule` instances are instantiated in Python, in this example, `TestModule(2.0)` and `TestModule(2)` create two instances with different types for its data attributes. `self.x` is of type `float` for `TestModule(2.0)`, whereas `self.y` is of type `int` for `TestModule(2.0)`.
- TorchScript automatically compiles other methods (e.g., `mul()`) invoked by methods annotated via `@torch.jit.export` or `forward()` methods.
- Entry-points to a TorchScript program are either `forward()` of a module type, functions annotated as `torch.jit.script`, or methods annotated as `torch.jit.export`.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]jit_language_reference_v2.rst, line 582)

Unknown directive type "testcode".

```
.. testcode::

    import torch
```

```
class TestModule(torch.nn.Module):
    def __init__(self, v):
        super().__init__()
        self.x = v

    def forward(self, inc: int):
        return self.x + inc

m = torch.jit.script(TestModule(1))
print(f"First instance: {m(3)}")

m = torch.jit.script(TestModule(torch.ones([5])))
print(f"Second instance: {m(3)}")
```

The example above produces the following output:

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]jit_language_reference_v2.rst`, line 602)

Unknown directive type "testoutput".

```
.. testoutput::

    First instance: 4
    Second instance: tensor([4., 4., 4., 4., 4.])
```

Example 2

The following example shows an incorrect usage of module type. Specifically, this example invokes the constructor of `TestModule` inside the scope of TorchScript:

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]jit_language_reference_v2.rst`, line 611)

Unknown directive type "testcode".

```
.. testcode::

    import torch

    class TestModule(torch.nn.Module):
        def __init__(self, v):
            super().__init__()
            self.x = v

        def forward(self, x: int):
            return self.x + x

    class MyModel:
        def __init__(self, v: int):
            self.val = v

        @torch.jit.export
        def doSomething(self, val: int) -> int:
            # error: should not invoke the constructor of module type
            myModel = TestModule(self.val)
            return myModel(val)

    # m = torch.jit.script(MyModel(2)) # Results in below RuntimeError
    # RuntimeError: Could not get name of python class object
```

Type Annotation

Since TorchScript is statically typed, programmers need to annotate types at *strategic points* of TorchScript code so that every local variable or instance data attribute has a static type, and every function and method has a statically typed signature.

When to Annotate Types

In general, type annotations are only needed in places where static types cannot be automatically inferred (e.g., parameters or sometimes return types to methods or functions). Types of local variables and data attributes are often automatically inferred from their assignment statements. Sometimes an inferred type may be too restrictive, e.g., `x` being inferred as `NoneType` through assignment `x = None`, whereas `x` is actually used as an `Optional`. In such cases, type annotations may be needed to overwrite auto inference, e.g., `x: Optional[int] = None`. Note that it is always safe to type annotate a local variable or data attribute even if its type can be automatically inferred. The annotated type must be congruent with TorchScriptâ€™s type-checking.

When a parameter, local variable, or data attribute is not type annotated and its type cannot be automatically inferred, TorchScript assumes it to be a default type of `TensorType`, `List[TensorType]`, or `Dict[str, TensorType]`.

Annotate Function Signature

Since a parameter may not be automatically inferred from the body of the function (including both functions and methods), they need to be type annotated. Otherwise, they assume the default type `TensorType`.

TorchScript supports two styles for method and function signature type annotation:

- **Python3-style** annotates types directly on the signature. As such, it allows individual parameters to be left unannotated (whose type will be the default type of `TensorType`), or allows the return type to be left unannotated (whose type will be automatically inferred).

```
Python3Annotation ::= "def" Identifier [ "(" ParamAnnot* ")" ] [ReturnAnnot] ":"
                            FuncOrMethodBody
ParamAnnot        ::= Identifier [ ":" TSType ] ","
ReturnAnnot       ::= "->" TSType
```

Note that when using Python3 style, the type `self` is automatically inferred and should not be annotated.

- **Mypy style** annotates types as a comment right below the function/method declaration. In the Mypy style, since parameter names do not appear in the annotation, all parameters have to be annotated.

```
MyPyAnnotation ::= "# type:" "(" ParamAnnot* ")" [ ReturnAnnot ]
ParamAnnot     ::= TSType ","
ReturnAnnot    ::= "->" TSType
```

Example 1

In this example:

- `a` is not annotated and assumes the default type of `TensorType`.
- `b` is annotated as type `int`.
- The return type is not annotated and is automatically inferred as type `TensorType` (based on the type of the value being returned).

```
import torch

def f(a, b: int):
    return a+b

m = torch.jit.script(f)
print("TorchScript:", m(torch.ones([6]), 100))
```

Example 2

The following example uses Mypy style annotation. Note that parameters or return values must be annotated even if some of them assume the default type.

```
import torch

def f(a, b):
    # type: (torch.Tensor, int) â†’ torch.Tensor
    return a+b

m = torch.jit.script(f)
print("TorchScript:", m(torch.ones([6]), 100))
```

Annotate Variables and Data Attributes

In general, types of data attributes (including class and instance data attributes) and local variables can be automatically inferred from assignment statements. Sometimes, however, if a variable or attribute is associated with values of different types (e.g., as `None` or `TensorType`), then they may need to be explicitly type annotated as a *wider* type such as `Optional[int]` or `Any`.

Local Variables

Local variables can be annotated according to Python3 typing module annotation rules, i.e.,

```
LocalVarAnnotation ::= Identifier [":" TSType] "=" Expr
```

In general, types of local variables can be automatically inferred. In some cases, however, you may need to annotate a multi-type for local variables that may be associated with different concrete types. Typical multi-types include `Optional[T]` and `Any`.

Example

```
import torch

def f(a, setVal: bool):
    value: Optional[torch.Tensor] = None
    if setVal:
        value = a
    return value

ones = torch.ones([6])
m = torch.jit.script(f)
print("TorchScript:", m(ones, True), m(ones, False))
```

Instance Data Attributes

For `ModuleType` classes, instance data attributes can be annotated according to Python3 typing module annotation rules. Instance data attributes can be annotated (optionally) as final via `Final`.

```
"class" ClassIdentifier "(torch.nn.Module):"
InstanceAttrIdentifier ":" ["Final("] TSType [")"]
...
```

Where:

- `InstanceAttrIdentifier` is the name of an instance attribute.
- `Final` indicates that the attribute cannot be re-assigned outside of `__init__` or overridden in subclasses.

Example

```
import torch

class MyModule(torch.nn.Module):
    offset_: int

def __init__(self, offset):
    self.offset_ = offset

...
```

Type Annotation APIs

`torch.jit.annotate(T, expr)`

This API annotates type `T` to an expression `expr`. This is often used when the default type of an expression is not the type intended by the programmer. For instance, an empty list (dictionary) has the default type of `List[TensorType]` (`Dict[TensorType, TensorType]`), but sometimes it may be used to initialize a list of some other types. Another common use case is for annotating the return type of `tensor.tolist()`. Note, however, that it cannot be used to annotate the type of a module attribute in `__init__`; `torch.jit.Attribute` should be used for this instead.

Example

In this example, `[]` is declared as a list of integers via `torch.jit.annotate` (instead of assuming `[]` to be the default type of `List[TensorType]`):

```
import torch
from typing import List

def g(l: List[int], val: int):
    l.append(val)
    return l

def f(val: int):
    l = g(torch.jit.annotate(List[int], []), val)
    return l

m = torch.jit.script(f)
print("Eager:", f(3))
print("TorchScript:", m(3))
```

See :meth:`torch.jit.annotate` for more information.

> **System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]jit_language_reference_v2.rst`, line 814); *backlink***
>
> Unknown interpreted text role "meth".

Type Annotation Appendix

TorchScript Type System Definition

```
TSAllType       ::= TSType | TSModuleType
TSType          ::= TSMetaType | TSPrimitiveType | TSStructuralType | TSNominalType

TSMetaType      ::= "Any"
TSPrimitiveType ::= "int" | "float" | "double" | "complex" | "bool" | "str" | "None"

TSStructualType ::=  TSTuple | TSNamedTuple | TSList | TSDict |
                     TSOptional | TSUnion | TSFuture | TSRRef
TSTuple         ::= "Tuple" "[" (TSType ",")* TSType "]"
TSNamedTuple    ::= "namedtuple" "(" (TSType ",")* TSType ")"
TSList          ::= "List" "[" TSType "]"
TSOptional      ::= "Optional" "[" TSType "]"
TSUnion         ::= "Union" "[" (TSType ",")* TSType "]"
TSFuture        ::= "Future" "[" TSType "]"
TSRRef          ::= "RRef" "[" TSType "]"
TSDict          ::= "Dict" "[" KeyType "," TSType "]"
KeyType         ::= "str" | "int" | "float" | "bool" | TensorType | "Any"

TSNominalType   ::= TSBuiltinClasses | TSCustomClass | TSEnum
TSBuiltinClass  ::= TSTensor | "torch.device" | "torch.stream"|
                    "torch.dtype" | "torch.nn.ModuleList" |
                    "torch.nn.ModuleDict" | ...
TSTensor        ::= "torch.tensor" and subclasses
```

Unsupported Typing Constructs

TorchScript does not support all features and types of the Python3 typing module. Any functionality from the typing module that is not explicitly specified in this documentation is unsupported. The following table summarizes `typing` constructs that are either unsupported or supported with restrictions in TorchScript.

Item	Description
`typing.Any`	In development
`typing.NoReturn`	Not supported

`typing.Callable`	Not supported
`typing.Literal`	Not supported
`typing.ClassVar`	Not supported
`typing.Final`	Supported for module attributes, class attribute, and annotations, but not for functions.
`typing.AnyStr`	Not supported
`typing.overload`	In development
Type aliases	Not supported
Nominal typing	In development
Structural typing	Not supported
NewType	Not supported
Generics	Not supported

Expressions

The following section describes the grammar of expressions that are supported in TorchScript. It is modeled after the expressions chapter of the Python language reference.

Arithmetic Conversions

There are a number of implicit type conversions that are performed in TorchScript:

- A `Tensor` with a `float` or `int` data type can be implicitly converted to an instance of `FloatType` or `IntType` provided that it has a size of 0, does not have `require_grad` set to `True`, and will not require narrowing.
- Instances of `StringType` can be implicitly converted to `DeviceType`.
- The implicit conversion rules from the two bullet points above can be applied to instances of `TupleType` to produce instances of `ListType` with the appropriate contained type.

Explicit conversions can be invoked using the `float`, `int`, `bool`, and `str` built-in functions that accept primitive data types as arguments and can accept user-defined types if they implement `__bool__`, `__str__`, etc.

Atoms

Atoms are the most basic elements of expressions.

```
atom      ::=  identifier | literal | enclosure
enclosure ::=  parenth_form | list_display | dict_display
```

Identifiers

The rules that dictate what is a legal identifer in TorchScript are the same as their Python counterparts.

Literals

```
literal ::=  stringliteral | integer | floatnumber
```

Evaluation of a literal yields an object of the appropriate type with the specific value (with approximations applied as necessary for floats). Literals are immutable, and multiple evaluations of identical literals may obtain the same object or distinct objects with the same value. stringliteral, integer, and floatnumber are defined in the same way as their Python counterparts.

Parenthesized Forms

```
parenth_form ::=  '(' [expression_list] ')'
```

A parenthesized expression list yields whatever the expression list yields. If the list contains at least one comma, it yields a `Tuple`; otherwise, it yields the single expression inside the expression list. An empty pair of parentheses yields an empty `Tuple` object (`Tuple[]`).

List and Dictionary Displays

```
list_comprehension ::=  expression comp_for
comp_for           ::=  'for' target_list 'in' or_expr
list_display       ::=  '[' [expression_list | list_comprehension] ']'
dict_display       ::=  '{' [key_datum_list | dict_comprehension] '}'
key_datum_list     ::=  key_datum (',' key_datum)*
key_datum          ::=  expression ':' expression
dict_comprehension ::=  key_datum comp_for
```

Lists and dicts can be constructed by either listing the container contents explicitly or by providing instructions on how to compute them via a set of looping instructions (i.e. a *comprehension*). A comprehension is semantically equivalent to using a for loop and appending to an ongoing list. Comprehensions implicitly create their own scope to make sure that the items of the target list do not leak into the enclosing scope. In the case that container items are explicitly listed, the expressions in the expression list are evaluated left-to-right. If a key is repeated in a `dict_display` that has a `key_datum_list`, the resultant dictionary uses the value from the rightmost datum in the list that uses the repeated key.

Primaries

```
primary ::=  atom | attributeref | subscription | slicing | call
```

Attribute References

```
attributeref ::=  primary '.' identifier
```

The `primary` must evaluate to an object of a type that supports attribute references that have an attribute named `identifier`.

Subscriptions

```
subscription ::=  primary '[' expression_list ']'
```

The `primary` must evaluate to an object that supports subscription.

- If the primary is a `List`, `Tuple`, or `str`, the expression list must evaluate to an integer or slice.
- If the primary is a `Dict`, the expression list must evaluate to an object of the same type as the key type of the `Dict`.
- If the primary is a `ModuleList`, the expression list must be an `integer` literal.
- If the primary is a `ModuleDict`, the expression must be a `stringliteral`.

Slicings

A slicing selects a range of items in a `str`, `Tuple`, `List`, or `Tensor`. Slicings may be used as expressions or targets in assignment or `del` statements.

```
slicing      ::=  primary '[' slice_list ']'
slice_list   ::=  slice_item (',' slice_item)* [',']
slice_item   ::=  expression | proper_slice
proper_slice ::=  [expression] ':' [expression] [':' [expression] ]
```

Slicings with more than one slice item in their slice lists can only be used with primaries that evaluate to an object of type `Tensor`.

Calls

```
call          ::=  primary '(' argument_list ')'
argument_list ::=  args [',' kwargs] | kwargs
args          ::=  [arg (',' arg)*]
kwargs        ::=  [kwarg (',' kwarg)*]
kwarg         ::=  arg '=' expression
arg           ::=  identifier
```

The `primary` must desugar or evaluate to a callable object. All argument expressions are evaluated before the call is attempted.

Power Operator

```
power ::=  primary ['**' u_expr]
```

The power operator has the same semantics as the built-in pow function (not supported); it computes its left argument raised to the power of its right argument. It binds more tightly than unary operators on the left, but less tightly than unary operators on the right; i.e. `-2 ** -3 == -(2 ** (-3))`. The left and right operands can be `int`, `float` or `Tensor`. Scalars are broadcast in the case of scalar-tensor/tensor-scalar exponentiation operations, and tensor-tensor exponentiation is done elementwise without any broadcasting.

Unary and Arithmetic Bitwise Operations

```
u_expr ::=  power | '-' power | '~' power
```

The unary `-` operator yields the negation of its argument. The unary `~` operator yields the bitwise inversion of its argument. `-` can be used with `int`, `float`, and `Tensor` of `int` and `float`. `~` can only be used with `int` and `Tensor` of `int`.

Binary Arithmetic Operations

```
m_expr ::=  u_expr | m_expr '*' u_expr | m_expr '@' m_expr | m_expr '//' u_expr | m_expr '/' u_expr | m_expr '%
a_expr ::=  m_expr | a_expr '+' m_expr | a_expr '-' m_expr
```

The binary arithmetic operators can operate on `Tensor`, `int`, and `float`. For tensor-tensor ops, both arguments must have the same shape. For scalar-tensor or tensor-scalar ops, the scalar is usually broadcast to the size of the tensor. Division ops can only accept scalars as their right-hand side argument, and do not support broadcasting. The `@` operator is for matrix multiplication and only operates on `Tensor` arguments. The multiplication operator (`*`) can be used with a list and integer in order to get a result that is the original list repeated a certain number of times.

Shifting Operations

```
shift_expr ::=  a_expr | shift_expr ( '<<' | '>>' ) a_expr
```

These operators accept two `int` arguments, two `Tensor` arguments, or a `Tensor` argument and an `int` or `float` argument. In all cases, a right shift by `n` is defined as floor division by `pow(2, n)`, and a left shift by `n` is defined as multiplication by `pow(2, n)`. When both arguments are `Tensors`, they must have the same shape. When one is a scalar and the other is a `Tensor`, the scalar is logically broadcast to match the size of the `Tensor`.

Binary Bitwise Operations

```
and_expr ::=  shift_expr | and_expr '&' shift_expr
xor_expr ::=  and_expr | xor_expr '^' and_expr
or_expr  ::=  xor_expr | or_expr '|' xor_expr
```

The `&` operator computes the bitwise AND of its arguments, the `^` the bitwise XOR, and the `|` the bitwise OR. Both operands must be `int` or `Tensor`, or the left operand must be `Tensor` and the right operand must be `int`. When both operands are `Tensor`, they must have the same shape. When the right operand is `int`, and the left operand is `Tensor`, the right operand is logically broadcast to match the shape of the `Tensor`.

Comparisons

```
comparison    ::=  or_expr (comp_operator or_expr)*
comp_operator ::=  '<' | '>' | '==' | '>=' | '<=' | '!=' | 'is' ['not'] | ['not'] 'in'
```

A comparison yields a boolean value (`True` or `False`), or if one of the operands is a `Tensor`, a boolean `Tensor`. Comparisons can be chained arbitrarily as long as they do not yield boolean `Tensors` that have more than one element. `a op1 b op2 c ...` is equivalent to `a op1 b and b op2 c and ...`.

Value Comparisons

The operators <, >, ==, >=, <=, and `!=` compare the values of two objects. The two objects generally need to be of the same type, unless there is an implicit type conversion available between the objects. User-defined types can be compared if rich comparison methods (e.g., `__lt__`) are defined on them. Built-in type comparison works like Python:

- Numbers are compared mathematically.
- Strings are compared lexicographically.
- `lists`, `tuples`, and `dicts` can be compared only to other `lists`, `tuples`, and `dicts` of the same type and are compared using the comparison operator of corresponding elements.

Membership Test Operations

The operators `in` and `not in` test for membership. `x in s` evaluates to `True` if `x` is a member of `s` and `False` otherwise. `x not in s` is equivalent to `not x in s`. This operator is supported for `lists`, `dicts`, and `tuples`, and can be used with user-defined types if they implement the `__contains__` method.

Identity Comparisons

For all types except `int`, `double`, `bool`, and `torch.device`, operators `is` and `is not` test for the objectâ€™s identity; `x is y` is `True` if and and only if `x` and `y` are the same object. For all other types, `is` is equivalent to comparing them using `==`. `x is not y` yields the inverse of `x is y`.

Boolean Operations

```
or_test  ::=  and_test | or_test 'or' and_test
and_test ::=  not_test | and_test 'and' not_test
not_test ::=  'bool' '(' or_expr ')' | comparison | 'not' not_test
```

User-defined objects can customize their conversion to `bool` by implementing a `__bool__` method. The operator `not` yields `True` if its operand is false, `False` otherwise. The expression `x` and `y` first evaluates `x`; if it is `False`, its value (`False`) is returned; otherwise, `y` is evaluated and its value is returned (`False` or `True`). The expression `x` or `y` first evaluates `x`; if it is `True`, its value (`True`) is returned; otherwise, `y` is evaluated and its value is returned (`False` or `True`).

Conditional Expressions

```
conditional_expression ::=  or_expr ['if' or_test 'else' conditional_expression]
 expression            ::=  conditional_expression
```

The expression `x if c else y` first evaluates the condition `c` rather than x. If `c` is `True`, `x` is evaluated and its value is returned; otherwise, `y` is evaluated and its value is returned. As with if-statements, `x` and `y` must evaluate to a value of the same type.

Expression Lists

```
expression_list ::=  expression (',' expression)* [',']
starred_item    ::=  '*' primary
```

A starred item can only appear on the left-hand side of an assignment statement, e.g., `a, *b, c = ...`.

Simple Statements

The following section describes the syntax of simple statements that are supported in TorchScript. It is modeled after the simple statements chapter of the Python language reference.

Expression Statements

```
expression_stmt    ::=  starred_expression
starred_expression ::=  expression | (starred_item ",")* [starred_item]
starred_item       ::=  assignment_expression | "*" or_expr
```

Assignment Statements

```
assignment_stmt ::=  (target_list "=")+ (starred_expression)
target_list     ::=  target ("," target)* [","]
target          ::=  identifier
                     | "(" [target_list] ")"
                     | "[" [target_list] "]"
                     | attributeref
                     | subscription
                     | slicing
                     | "*" target
```

Augmented Assignment Statements

```
augmented_assignment_stmt ::= augtarget augop (expression_list)
```

```
augtarget                 ::= identifier | attributeref | subscription
augop                     ::= "+=" | "-=" | "*=" | "/=" | "//=" | "%=" |
                              "**="| ">>=" | "<<=" | "&=" | "^=" | "|="
```

Annotated Assignment Statements

```
annotated_assignment_stmt ::= augtarget ":" expression
                              ["=" (starred_expression)]
```

The `raise` Statement

```
raise_stmt ::=  "raise" [expression ["from" expression]]
```

Raise statements in TorchScript do not support `try\except\finally`.

The `assert` Statement

```
assert_stmt ::=  "assert" expression ["," expression]
```

Assert statements in TorchScript do not support `try\except\finally`.

The `return` Statement

```
return_stmt ::=  "return" [expression_list]
```

Return statements in TorchScript do not support `try\except\finally`.

The `del` Statement

```
del_stmt ::=  "del" target_list
```

The `pass` Statement

```
pass_stmt ::= "pass"
```

The `print` Statement

```
print_stmt ::= "print" "(" expression  [, expression] [.format{expression_list}] ")"
```

The `break` Statement

```
break_stmt ::= "break"
```

The `continue` Statement:

```
continue_stmt ::= "continue"
```

Compound Statements

The following section describes the syntax of compound statements that are supported in TorchScript. The section also highlights how Torchscript differs from regular Python statements. It is modeled after the compound statements chapter of the Python language reference.

The `if` Statement

Torchscript supports both basic `if/else` and ternary `if/else`.

Basic `if/else` Statement

```
if_stmt ::= "if" assignment_expression ":" suite
            ("elif" assignment_expression ":" suite)
            ["else" ":" suite]
```

`elif` statements can repeat for an arbitrary number of times, but it needs to be before `else` statement.

Ternary `if/else` Statement

```
if_stmt ::= return [expression_list] "if" assignment_expression "else" [expression_list]
```

Example 1

A `tensor` with 1 dimension is promoted to `bool`:

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]jit_language_reference_v2.rst`, line 1313)

Unknown directive type "testcode".

```
.. testcode::

    import torch

    @torch.jit.script
    def fn(x: torch.Tensor):
        if x: # The tensor gets promoted to bool
            return True
```

```
        return False
    print(fn(torch.rand(1)))
```

The example above produces the following output:

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]jit_language_reference_v2.rst`, line 1326)

Unknown directive type "testoutput".

```
.. testoutput::

    True
```

Example 2

A `tensor` with multi dimensions are not promoted to `bool`:

```
import torch

# Multi dimensional Tensors error out.

@torch.jit.script
def fn():
    if torch.rand(2):
        print("Tensor is available")

    if torch.rand(4,5,6):
        print("Tensor is available")

print(fn())
```

Running the above code yields the following `RuntimeError`.

```
RuntimeError: The following operation failed in the TorchScript interpreter.
Traceback of TorchScript (most recent call last):
@torch.jit.script
def fn():
    if torch.rand(2):
       ~~~~~~~~~~~~ <--- HERE
        print("Tensor is available")
RuntimeError: Boolean value of Tensor with more than one value is ambiguous
```

If a conditional variable is annotated as `final`, either the true or false branch is evaluated depending on the evaluation of the conditional variable.

Example 3

In this example, only the True branch is evaluated, since `a` is annotated as `final` and set to `True`:

```
import torch

a : torch.jit.final[Bool] = True

if a:
    return torch.empty(2,3)
else:
    return []
```

The `while` Statement

```
while_stmt ::=  "while" assignment_expression ":" suite
```

while...else statements are not supported in Torchscript. It results in a `RuntimeError`.

The `for-in` Statement

```
for_stmt ::=  "for" target_list "in" expression_list ":" suite
              ["else" ":" suite]
```

`for...else` statements are not supported in Torchscript. It results in a `RuntimeError`.

Example 1

For loops on tuples: these unroll the loop, generating a body for each member of the tuple. The body must type-check correctly for each member.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]jit_language_reference_v2.rst`, line 1404)

Unknown directive type "testcode".

```
.. testcode::

    import torch
    from typing import Tuple

    @torch.jit.script
    def fn():
```

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

    fn()
```

The example above produces the following output:

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]jit_language_reference_v2.rst`, line 1419)

Unknown directive type "testoutput".

```
.. testoutput::

    3
     1
     1
     1
     1
    [ CPUFloatType{4} ]
```

Example 2

For loops on lists: for loops over a `nn.ModuleList` will unroll the body of the loop at compile time, with each member of the module list.

```
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):

    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

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

The `with` Statement

The `with` statement is used to wrap the execution of a block with methods defined by a context manager.

```
with_stmt ::=  "with" with_item ("," with_item) ":" suite
with_item ::=  expression ["as" target]
```

- If a target was included in the `with` statement, the return value from the context managerâ€™s `__enter__()` is assigned to it. Unlike python, if an exception caused the suite to be exited, its type, value, and traceback are not passed as arguments to `__exit__()`. Three `None` arguments are supplied.
- `try`, `except`, and `finally` statements are not supported inside `with` blocks.
- Exceptions raised within `with` block cannot be suppressed.

The `tuple` Statement

```
tuple_stmt ::= tuple([iterables])
```

- Iterable types in TorchScript include `Tensors`, `lists`, `tuples`, `dictionaries`, `strings`, `torch.nn.ModuleList`, and `torch.nn.ModuleDict`.
- You cannot convert a List to Tuple by using this built-in function.

Unpacking all outputs into a tuple is covered by:

```
abc = func() # Function that returns a tuple
a,b = func()
```

The `getattr` Statement

```
getattr_stmt ::= getattr(object, name[, default])
```

- Attribute name must be a literal string.
- Module type object is not supported (e.g., torch._C).
- Custom class object is not supported (e.g., torch.classes.*).

The `hasattr` Statement

```
hasattr_stmt ::= hasattr(object, name)
```

- Attribute name must be a literal string.
- Module type object is not supported (e.g., torch._C).
- Custom class object is not supported (e.g., torch.classes.*).

The `zip` Statement

```
zip_stmt ::= zip(iterable1, iterable2)
```

- Arguments must be iterables.
- Two iterables of same outer container type but different length are supported.

Example 1

Both the iterables must be of the same container type:

> **System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]jit_language_reference_v2.rst`, line 1522)**
>
> Unknown directive type "testcode".
>
> ```
> .. testcode::
>
>     a = [1, 2] # List
>     b = [2, 3, 4] # List
>     zip(a, b) # works
> ```

Example 2

This example fails because the iterables are of different container types:

```
a = (1, 2) # Tuple
b = [2, 3, 4] # List
zip(a, b) # Runtime error
```

Running the above code yields the following `RuntimeError`.

```
RuntimeError: Can not iterate over a module list or
    tuple with a value that does not have a statically determinable length.
```

Example 3

Two iterables of the same container Type but different data type is supported:

> **System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]jit_language_reference_v2.rst`, line 1549)**
>
> Unknown directive type "testcode".
>
> ```
> .. testcode::
>
>     a = [1.3, 2.4]
>     b = [2, 3, 4]
>     zip(a, b) # Works
> ```

Iterable types in TorchScript include `Tensors`, `lists`, `tuples`, `dictionaries`, `strings`, `torch.nn.ModuleList`, and `torch.nn.ModuleDict`.

The `enumerate` Statement

```
enumerate_stmt ::= enumerate([iterable])
```

- Arguments must be iterables.
- Iterable types in TorchScript include `Tensors`, `lists`, `tuples`, `dictionaries`, `strings`, `torch.nn.ModuleList` and `torch.nn.ModuleDict`.

Python Values

Resolution Rules

When given a Python value, TorchScript attempts to resolve it in the following five different ways:

- Compilable Python Implementation:
  - When a Python value is backed by a Python implementation that can be compiled by TorchScript, TorchScript compiles and uses the underlying Python implementation.
  - Example: `torch.jit.Attribute`
- Op Python Wrapper:
  - When a Python value is a wrapper of a native PyTorch op, TorchScript emits the corresponding operator.
  - Example: `torch.jit._logging.add_stat_value`
- Python Object Identity Match:
  - For a limited set of `torch.*` API calls (in the form of Python values) that TorchScript supports, TorchScript attempts to match a Python value against each item in the set.
  - When matched, TorchScript generates a corresponding `SugaredValue` instance that contains lowering logic for

these values.
    - Example: `torch.jit.isinstance()`

- Name Match:
    - For Python built-in functions and constants, TorchScript identifies them by name, and creates a corresponding `SugaredValue` instance that implements their functionality.
    - Example: `all()`

- Value Snapshot:
    - For Python values from unrecognized modules, TorchScript attempts to take a snapshot of the value and converts it to a constant in the graph of the function(s) or method(s) that are being compiled.
    - Example: `math.pi`

Python Built-in Functions Support

TorchScript Support for Python Built-in Functions

Built-in Function	Support Level	Notes
`abs()`	Partial	Only supports `Tensor`/`Int`/`Float` type inputs. | Doesn't honor `__abs__` override.
`all()`	Full	
`any()`	Full	
`ascii()`	None	
`bin()`	Partial	Only supports `Int` type input.
`bool()`	Partial	Only supports `Tensor`/`Int`/`Float` type inputs.
`breakpoint()`	None	
`bytearray()`	None	
`bytes()`	None	
`callable()`	None	
`chr()`	Partial	Only ASCII character set is supported.
`classmethod()`	Full	
`compile()`	None	
`complex()`	None	
`delattr()`	None	
`dict()`	Full	
`dir()`	None	
`divmod()`	Full	
`enumerate()`	Full	
`eval()`	None	
`exec()`	None	
`filter()`	None	
`float()`	Partial	Doesn't honor `__index__` override.
`format()`	Partial	Manual index specification not supported. | Format type modifier not supported.
`frozenset()`	None	
`getattr()`	Partial	Attribute name must be string literal.
`globals()`	None	
`hasattr()`	Partial	Attribute name must be string literal.
`hash()`	Full	`Tensor`'s hash is based on identity not numeric value.
`hex()`	Partial	Only supports `Int` type input.
`id()`	Full	Only supports `Int` type input.
`input()`	None	
`int()`	Partial	`base` argument not supported. | Doesn't honor `__index__` override.
`isinstance()`	Full	`torch.jit.isintance` provides better support when checking against container types like `Dict[str, int]`.
`issubclass()`	None	
`iter()`	None	
`len()`	Full	
`list()`	Full	
`ord()`	Partial	Only ASCII character set is supported.
`pow()`	Full	
`print()`	Partial	`separate`, `end` and `file` arguments are not supported.
`property()`	None	
`range()`	Full	
`repr()`	None	
`reversed()`	None	
`round()`	Partial	`ndigits` argument is not supported.
`set()`	None	
`setattr()`	None	
`slice()`	Full	
`sorted()`	Partial	`key` argument is not supported.
`staticmethod()`	Full	

Built-in Function	Support Level	Notes
`str()`	Partial	`encoding` and `errors` arguments are not supported.
`sum()`	Full	
`super()`	Partial	It can only be used in `nn.Module`'s `__init__` method.
`type()`	None	
`vars()`	None	
`zip()`	Full	
`__import__()`	None	

Python Built-in Values Support

TorchScript Support for Python Built-in Values

Built-in Value	Support Level	Notes
`False`	Full	
`True`	Full	
`None`	Full	
`NotImplemented`	None	
`Ellipsis`	Full	

torch.* APIs

Remote Procedure Calls

TorchScript supports a subset of RPC APIs that supports running a function on a specified remote worker instead of locally.

Specifically, following APIs are fully supported:

- `torch.distributed.rpc.rpc_sync()`
  - `rpc_sync()` makes a blocking RPC call to run a function on a remote worker. RPC messages are sent and received in parallel to execution of Python code.
  - More details about its usage and examples can be found in :meth:`~torch.distributed.rpc.rpc_sync`.

    System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]jit_language_reference_v2.rst`, line 1829); *backlink*

    Unknown interpreted text role "meth".

- `torch.distributed.rpc.rpc_async()`
  - `rpc_async()` makes a non-blocking RPC call to run a function on a remote worker. RPC messages are sent and received in parallel to execution of Python code.
  - More deatils about its usage and examples can be found in :meth:`~torch.distributed.rpc.rpc_async`.

    System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]jit_language_reference_v2.rst`, line 1833); *backlink*

    Unknown interpreted text role "meth".

- `torch.distributed.rpc.remote()`
  - `remote.()` executes a remote call on a worker and gets a Remote Reference `RRef` as the return value.
  - More details about its usage and examples can be found in :meth:`~torch.distributed.rpc.remote`.

    System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]jit_language_reference_v2.rst`, line 1836); *backlink*

    Unknown interpreted text role "meth".

Asynchronous Execution

TorchScript enables you to create asynchronous computation tasks to make better use of computation resources. This is done via supporting a list of APIs that are only usable within TorchScript:

- `torch.jit.fork()`
  - Creates an asynchronous task executing func and a reference to the value of the result of this execution. Fork will return immediately.
  - Synonymous to `torch.jit._fork()`, which is only kept for backward compatibility reasons.
  - More deatils about its usage and examples can be found in :meth:`~torch.jit.fork`.

    System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs]` [source]jit_language_reference_v2.rst, line 1850); *backlink*

Unknown interpreted text role "meth".

- `torch.jit.wait()`
    - Forces completion of a `torch.jit.Future[T]` asynchronous task, returning the result of the task.
    - Synonymous to `torch.jit._wait()`, which is only kept for backward compatibility reasons.
    - More deatils about its usage and examples can be found in :meth:`~torch.jit.wait`.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]jit_language_reference_v2.rst, line 1854); *backlink*

Unknown interpreted text role "meth".

Type Annotations

TorchScript is statically-typed. It provides and supports a set of utilities to help annotate variables and attributes:

- `torch.jit.annotate()`
    - Provides a type hint to TorchScript where Python 3 style type hints do not work well.
    - One common example is to annotate type for expressions like `[]`. `[]` is treated as `List[torch.Tensor]` by default. When a different type is needed, you can use this code to hint TorchScript: `torch.jit.annotate(List[int], [])`.
    - More details can be found in :meth:`~torch.jit.annotate`

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]jit_language_reference_v2.rst, line 1867); *backlink*

Unknown interpreted text role "meth".

- `torch.jit.Attribute`
    - Common use cases include providing type hint for `torch.nn.Module` attributes. Because their `__init__` methods are not parsed by TorchScript, `torch.jit.Attribute` should be used instead of `torch.jit.annotate` in the module's `__init__` methods.
    - More details can be found in :meth:`~torch.jit.Attribute`

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]jit_language_reference_v2.rst, line 1870); *backlink*

Unknown interpreted text role "meth".

- `torch.jit.Final`
    - An alias for Python's `typing.Final`. `torch.jit.Final` is kept only for backward compatibility reasons.

Meta Programming

TorchScript provides a set of utilities to facilitate meta programming:

- `torch.jit.is_scripting()`
    - Returns a boolean value indicating whether the current program is compiled by `torch.jit.script` or not.
    - When used in an `assert` or an `if` statement, the scope or branch where `torch.jit.is_scripting()` evaluates to `False` is not compiled.
    - Its value can be evaluated statically at compile time, thus commonly used in `if` statements to stop TorchScript from compiling one of the branches.
    - More details and examples can be found in :meth:`~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_v2.rst, line 1886); *backlink*

Unknown interpreted text role "meth".

- `torch.jit.is_tracing()`
    - Returns a boolean value indicating whether the current program is traced by `torch.jit.trace` / `torch.jit.trace_module` or not.
    - More details can be found in :meth:`~torch.jit.is_tracing`

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]jit_language_reference_v2.rst, line 1889); *backlink*

Unknown interpreted text role "meth".

- `@torch.jit.ignore`
  - This decorator indicates to the compiler that a function or method should be ignored and left as a Python function.
  - This allows you to leave code in your model that is not yet TorchScript compatible.
  - If a function decorated by `@torch.jit.ignore` is called from TorchScript, ignored functions will dispatch the call to the Python interpreter.
  - Models with ignored functions cannot be exported.
  - More details and examples can be found in :meth:`~torch.jit.ignore`

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]jit_language_reference_v2.rst, line 1895); *backlink*

Unknown interpreted text role "meth".

- `@torch.jit.unused`
  - This decorator indicates to the compiler that a function or method should be ignored and replaced with the raising of an exception.
  - This allows you to leave code in your model that is not yet TorchScript compatible and still export your model.
  - If a function decorated by `@torch.jit.unused` is called from TorchScript, a runtime error will be raised.
  - More details and examples can be found in :meth:`~torch.jit.unused`

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]jit_language_reference_v2.rst, line 1900); *backlink*

Unknown interpreted text role "meth".

Type Refinement

- `torch.jit.isinstance()`
  - Returns a boolean indicating whether a variable is of the specified type.
  - More details about its usage and examples can be found in :meth:`~torch.jit.isinstance`.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\[pytorch-master][docs][source]jit_language_reference_v2.rst, line 1909); *backlink*

Unknown interpreted text role "meth".