Python Mapping for Sequences

On this page:

Default Sequence Mapping in Python

A Slice sequence maps by default to a Python list; the only exception is a sequence of bytes, which maps by default to a string in order to lower memory utilization and improve throughput. This use of native types means that the Python mapping does not generate a separate named type for a Slice sequence. It also means that you can take advantage of all the inherent functionality offered by Python's native types. For example, here is the definition of our `FruitPlatter` sequence once more:

Python

```
sequence<Fruit> FruitPlatter;
```

We can use the `FruitPlatter` sequence as shown below:

Python

```
platter = [ Fruit.Apple, Fruit.Pear ]
assert(len(platter) == 2)
platter.append(Fruit.Orange)
```

The Ice run time validates the elements of a tuple or list to ensure that they are compatible with the declared type; a `ValueError` exception is raised if an incompatible type is encountered.

Allowable Sequence Values in Python

Although each sequence type has a default mapping, the Ice run time allows a sender to use other types as well. Specifically, a tuple is also accepted for a sequence type that maps to a list, and in the case of a byte sequence, the sender is allowed to supply a tuple or list of integers as an alternative to a string.

Using a string for a byte sequence bypasses the validation step and avoids an extra copy, resulting in much greater throughput than a tuple or list. For larger byte sequences, the use of a string is strongly recommended.

Furthermore, the Ice run time accepts objects that implement Python's buffer protocol as legal values for sequences of all primitive types except strings. For example, you can use the `array` module to create a buffer that is transferred much more efficiently than a tuple or list. Consider the two sequence values in the sample code below:

Python

```
import array
...
seq1 = array.array("i", [1, 2, 3, 4, 5])
seq2 = [1, 2, 3, 4, 5]
```

The values have the same on-the-wire representation, but they differ greatly in marshaling overhead because the buffer can be traversed more quickly and requires no validation.

Note that the Ice run time has no way of knowing what type of elements a buffer contains, therefore it is the application's responsibility to ensure that a buffer is compatible with the declared sequence type.

Customizing the Sequence Mapping in Python

The previous section described the allowable types that an application may use when sending a sequence. That kind of flexibility is not possible when receiving a sequence, because in this case it is the Ice run time's responsibility to create the container that holds the sequence.

As stated earlier, the default mapping for most sequence types is a list, and for byte sequences the default mapping is a string. Unless otherwise indicated, an application always receives sequences as the container type specified by the default mapping. If it would be more convenient to receive a sequence as a different type, you can customize the mapping by annotating your Slice definitions with metadata. The following table describes the metadata directives supported by the Python mapping:

Directive	Description
`python:seq:default`	Use the default mapping.
`python:seq:list`	Map to a Python list.
`python:seq:tuple`	Map to a Python tuple.

A metadata directive may be specified when defining a sequence, or when a sequence is used as a parameter, return value or data member. If specified at the point of definition, the directive affects all occurrences of that sequence type unless overridden by another directive at a point of use. The following Slice definitions illustrate these points:

Slice

```
sequence<int> IntList; // Uses list by default
["python:seq:tuple"] sequence<int> IntTuple; // Defaults to tuple

sequence<byte> ByteString; // Uses string by default
["python:seq:list"] sequence<byte> ByteList; // Defaults to list

struct S {
    IntList i1; // list
    IntTuple i2; // tuple
    ["python:seq:tuple"] IntList i3; // tuple
    ["python:seq:list"] IntTuple i4; // list
    ["python:seq:default"] IntTuple i5; // list

    ByteString b1; // string
    ByteList b2; // list
    ["python:seq:list"] ByteString b3; // list
    ["python:seq:tuple"] ByteString b4; // tuple
    ["python:seq:default"] ByteList b5; // string
};

interface I {
    IntList op1(ByteString s1, out ByteList s2);

    ["python:seq:tuple"]
    IntList op2(["python:seq:list"] ByteString s1,
                ["python:seq:tuple"] out ByteList s2);
};
```

The operation `op2` and the data members of structure `S` demonstrate how to override the mapping for a sequence at the point of use.

It is important to remember that these metadata directives only affect the receiver of the sequence. For example, the data members of structure `S` are populated with the specified sequence types only when the Ice run time unmarshals an instance of `S`. In the case of an operation, custom metadata affects the client when specified for the operation's return type and output parameters, whereas metadata affects the server for input parameters.

See Also

- Sequences
- Python Mapping for Identifiers
- Python Mapping for Modules
- Python Mapping for Built-In Types
- Python Mapping for Enumerations
- Python Mapping for Structures
- Python Mapping for Dictionaries