Python Mapping for Classes

On this page:

- Basic Python Mapping for Classes
- Inheritance from Ice.Object in Python
- Class Data Members in Python
- Class Constructors in Python
- Class Operations in Python
- Class Factories in Python

Basic Python Mapping for Classes

A Slice class maps to a Python class with the same name. The generated class contains an attribute for each Slice data member (just as for structures and exceptions). Consider the following class definition:

Slice

```
class TimeOfDay {
    short hour;         // 0 - 23
    short minute;       // 0 - 59
    short second;       // 0 - 59
    string format();    // Return time as hh:mm:ss
};
```

The Python mapping generates the following code for this definition:

Python

```
class TimeOfDay(Ice.Object):
    def __init__(self, hour=0, minute=0, second=0):
        # ...
        self.hour = hour
        self.minute = minute
        self.second = second

    def ice_staticId():
        return '::M::TimeOfDay'
    ice_staticId = staticmethod(ice_staticId)

    # ...

    #
    # Operation signatures.
    #
    # def format(self, current=None):
```

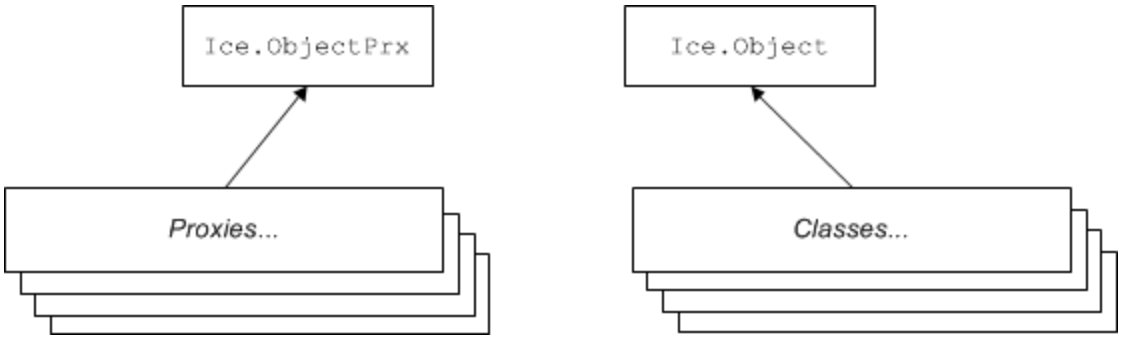
There are a number of things to note about the generated code:

1. The generated class `TimeOfDay` inherits from `Ice.Object`. This means that all classes implicitly inherit from `Ice.Object`, which is the ultimate ancestor of all classes. Note that `Ice.Object` is *not* the same as `Ice.ObjectPrx`. In other words, you *cannot* pass a class where a proxy is expected and vice versa.
2. The constructor defines an attribute for each Slice data member.
3. The class defines the static method `ice_staticId`.
4. A comment summarizes the method signatures for each Slice operation.

There is quite a bit to discuss here, so we will look at each item in turn.

Inheritance from `Ice.Object` in Python

Like interfaces, classes implicitly inherit from a common base class, `Ice.Object`. However, as shown in the illustration below, classes inherit from `Ice.Object` instead of `Ice.ObjectPrx` (which is at the base of the inheritance hierarchy for proxies). As a result, you cannot pass a class where a proxy is expected (and vice versa) because the base types for classes and proxies are not compatible.



Inheritance from `Ice.ObjectPrx` and `Ice.Object`.

`Ice.Object` contains a number of methods:

Python

```
class Object(object):
    def ice_isA(self, id, current=None):
        # ...

    def ice_ping(self, current=None):
        # ...

    def ice_ids(self, current=None):
        # ...

    def ice_id(self, current=None):
        # ...

    def ice_staticId():
        # ...
    ice_staticId = staticmethod(ice_staticId)

    def ice_preMarshal(self):
        # ...

    def ice_postUnmarshal(self):
        # ...
```

The member functions of `Ice.Object` behave as follows:

- `ice_isA`
  This method returns `true` if the object supports the given type ID, and `false` otherwise.
- `ice_ping`
  As for interfaces, `ice_ping` provides a basic reachability test for the object.
- `ice_ids`
  This method returns a string sequence representing all of the type IDs supported by this object, including `::Ice::Object`.
- `ice_id`
  This method returns the actual run-time type ID of the object. If you call `ice_id` through a reference to a base instance, the returned type ID is the actual (possibly more derived) type ID of the instance.
- `ice_staticId`
  This method is generated in each class and returns the static type ID of the class.
- `ice_preMarshal`
  The Ice run time invokes this method prior to marshaling the object's state, providing the opportunity for a subclass to validate its declared data members.

- `ice_postUnmarshal`
  The Ice run time invokes this method after unmarshaling an object's state. A subclass typically overrides this function when it needs to perform additional initialization using the values of its declared data members.

Note that neither `Ice.Object` nor the generated class override `__hash__` and `__eq__`, so the default implementations apply.

Class Data Members in Python

By default, data members of classes are mapped exactly as for structures and exceptions: for each data member in the Slice definition, the generated class contains a corresponding attribute.

Optional data members use the same mapping as required data members, but an optional data member can also be set to the marker value `Ice.Unset` to indicate that the member is unset. A well-behaved program must compare an optional data member to `Ice.Unset` before using the member's value:

Python

```
v = ...
if v.optionalMember is Ice.Unset:
    print("optionalMember is unset")
else:
    print("optionalMember = " + str(v.optionalMember))
```

Although Python provides no standard mechanism for restricting access to an object's attributes, by convention an attribute whose name begins with an underscore signals the author's intent that the attribute should only be accessed by the class itself or by one of its subclasses. You can employ this convention in your Slice classes using the `protected` metadata directive. The presence of this directive causes the Slice compiler to prepend an underscore to the mapped name of the data member. For example, the `TimeOfDay` class shown below has the `protected` metadata directive applied to each of its data members:

Slice

```
class TimeOfDay {
    ["protected"] short hour;   // 0 - 23
    ["protected"] short minute; // 0 - 59
    ["protected"] short second; // 0 - 59
    string format();    // Return time as hh:mm:ss
};
```

The Slice compiler produces the following generated code for this definition:

Python

```
class TimeOfDay(Ice.Object):
    def __init__(self, hour=0, minute=0, second=0):
        # ...
        self._hour = hour
        self._minute = minute
        self._second = second

    # ...

    #
    # Operation signatures.
    #
    # def format(self, current=None):
```

For a class in which all of the data members are protected, the metadata directive can be applied to the class itself rather than to each member individually. For example, we can rewrite the `TimeOfDay` class as follows:

Slice

```
["protected"] class TimeOfDay {
    short hour;         // 0 - 23
    short minute;       // 0 - 59
    short second;       // 0 - 59
    string format();    // Return time as hh:mm:ss
};
```

Class Constructors in Python

Classes have a constructor that assigns to each data member a default value appropriate for its type. You can also declare different default values for data members of primitive and enumerated types.

For derived classes, the constructor has one parameter for each of the base class's data members, plus one parameter for each of the derived class's data members, in base-to-derived order.

You can invoke this constructor in one of two ways:

- Provide values for all members, including optional members, in the order of declaration:

  Python

  ```
  t = TimeOfDay(12, 33, 45)
  t2 = TimeOfDay(14, 7) # second defaults to 0
  ```

  Pass `Ice.Unset` as the value of any optional member you want to be unset.

- Used named arguments to specify values for certain members and in any order:

  Python

  ```
  t = TimeOfDay(minute=33, hour=12)
  ```

Class Operations in Python

Operations of classes are mapped to methods in the generated class. This means that, if a class contains operations (such as the `format` operation of our `TimeOfDay` class), you must provide an implementation of the operation in a class that is derived from the generated class. For example:

Python

```
class TimeOfDayI(TimeOfDay):
    def __init__(self, hour=0, minute=0, second=0):
        TimeOfDay.__init__(self, hour, minute, second)

    def format(self, current=None):
        return "%02d:%02d:%02d" % (self.hour, self.minute, self.second)
```

A Slice class such as `TimeOfDay` that declares or inherits an operation is inherently abstract. Python does not support the notion of abstract classes or abstract methods, therefore the mapping merely summarizes the required method signatures in a comment for your convenience. Furthermore, the mapping generates code in the constructor of an abstract class to prevent it from being instantiated directly; any attempt to do so raises a `RuntimeError` exception.

You may notice that the mapping for an operation adds an optional trailing parameter named `current`. For now, you can ignore this parameter and pretend it does not exist.

Class Factories in Python

Having created a class such as `TimeOfDayI`, we have an implementation and we can instantiate the `TimeOfDayI` class, but we cannot receive it as the return value or as an out-parameter from an operation invocation. To see why, consider the following simple interface:

Slice

```
interface Time {
    TimeOfDay get();
};
```

When a client invokes the `get` operation, the Ice run time must instantiate and return an instance of the `TimeOfDay` class. However, `TimeOfDay` is an abstract class that cannot be instantiated. Unless we tell it, the Ice run time cannot magically know that we have created a `TimeOfDayI` class that implements the abstract `format` operation of the `TimeOfDay` abstract class. In other words, we must provide the Ice run time with a factory that knows that the `TimeOfDay` abstract class has a `TimeOfDayI` concrete implementation. The `Ice::Communicator` interface provides us with the necessary operations:

Slice

```
module Ice {
    local interface ObjectFactory {
        Object create(string type);
        void destroy();
    };

    local interface Communicator {
        void addObjectFactory(ObjectFactory factory, string id);
        ObjectFactory findObjectFactory(string id);
        // ...
    };
};
```

To supply the Ice run time with a factory for our `TimeOfDayI` class, we must implement the `ObjectFactory` interface:

Python

```
class ObjectFactory(Ice.ObjectFactory):
    def create(self, type):
        if type == M.TimeOfDay.ice_staticId():
            return TimeOfDayI()
        assert(False)
        return None

    def destroy(self):
        # Nothing to do
        pass
```

The object factory's `create` method is called by the Ice run time when it needs to instantiate a `TimeOfDay` class. The factory's `destroy` method is called by the Ice run time when its communicator is destroyed.

The `create` method is passed the type ID of the class to instantiate. For our `TimeOfDay` class, the type ID is `"::M::TimeOfDay"`. Our implementation of `create` checks the type ID: if it matches, the method instantiates and returns a `TimeOfDayI` object. For other type IDs, the method asserts because it does not know how to instantiate other types of objects.

Note that we used the `ice_staticId` method to obtain the type ID rather than embedding a literal string. Using a literal type ID string in your code is discouraged because it can lead to errors that are only detected at run time. For example, if a Slice class or one of its enclosing modules is renamed and the literal string is not changed accordingly, a receiver will fail to unmarshal the object and the Ice run time will raise `NoObjectFactoryException`. By using `ice_staticId` instead, we avoid any risk of a misspelled or obsolete type ID, and we can discover earlier whether a Slice class or module has been renamed.

Given a factory implementation, such as our `ObjectFactory`, we must inform the Ice run time of the existence of the factory:

Python

```
ic = ...   # Get Communicator...
ic.addObjectFactory(ObjectFactory(), M.TimeOfDay.ice_staticId())
```

Now, whenever the Ice run time needs to instantiate a class with the type ID `"::M::TimeOfDay"`, it calls the `create` method of the registered `ObjectFactory` instance.

The `destroy` operation of the object factory is invoked by the Ice run time when the communicator is destroyed. This gives you a chance to clean up any resources that may be used by your factory. Do not call `destroy` on the factory while it is registered with the communicator — if you do, the Ice run time has no idea that this has happened and, depending on what your `destroy` implementation is doing, may cause undefined behavior when the Ice run time tries to next use the factory.

The run time guarantees that `destroy` will be the last call made on the factory, that is, `create` will not be called concurrently with `destroy`, and `create` will not be called once `destroy` has been called. However, calls to `create` can be made concurrently.

Note that you cannot register a factory for the same type ID twice: if you call `addObjectFactory` with a type ID for which a factory is registered, the Ice run time throws an `AlreadyRegisteredException`.

Finally, keep in mind that if a class has only data members, but no operations, you need not create and register an object factory to transmit instances of such a class. Only if a class has operations do you have to define and register an object factory.

See Also

- Classes
- Type IDs
- Optional Data Members
- Python Mapping for Operations
- The Current Object