Code Generation in Python

The Python mapping supports two forms of code generation: dynamic and static.

On this page:

- Dynamic Code Generation in Python
  - Ice.loadSlice Options in Python
  - Locating Slice Files in Python
  - Loading Multiple Slice Files in Python
- Static Code Generation in Python
  - Compiler Output in Python
  - Include Files in Python
- Static Versus Dynamic Code Generation in Python
  - Application Considerations for Code Generation in Python
  - Mixing Static and Dynamic Code Generation in Python
- slice2py Command-Line Options
- Generating Packages in Python

Dynamic Code Generation in Python

Using dynamic code generation, Slice files are "loaded" at run time and dynamically translated into Python code, which is immediately compiled and available for use by the application. This is accomplished using the `Ice.loadSlice` function, as shown in the following example:

Python

```
Ice.loadSlice("Color.ice")
import M

print "My favorite color is", M.Color.blue
```

For this example, we assume that `Color.ice` contains the following definitions:

Slice

```
module M {
    enum Color { red, green, blue };
};
```

The code imports module `M` after the Slice file is loaded because module `M` is not defined until the Slice definitions have been translated into Python.

`Ice.loadSlice` Options in Python

The `Ice.loadSlice` function behaves like a Slice compiler in that it accepts command-line arguments for specifying preprocessor options and controlling code generation. The arguments must include at least one Slice file.

The function has the following Python definition:

Python

```
def Ice.loadSlice(cmd, args=[])
```

The command-line arguments can be specified entirely in the first argument, `cmd`, which must be a string. The optional second argument can be used to pass additional command-line arguments as a list; this is useful when the caller already has the arguments in list form. The function always returns `None`.

For example, the following calls to `Ice.loadSlice` are functionally equivalent:

Python

```
Ice.loadSlice("-I/opt/IcePy/slice Color.ice")
Ice.loadSlice("-I/opt/IcePy/slice", ["Color.ice"])
Ice.loadSlice("", ["-I/opt/IcePy/slice", "Color.ice"])
```

In addition to the standard compiler options, `Ice.loadSlice` also supports the following command-line options:

- `--all`
  Generate code for all Slice definitions, including those from included files.
- `--checksum`
  Generate checksums for Slice definitions.

Locating Slice Files in Python

If your Slice files depend on Ice types, you can avoid hard-coding the path name of your Ice installation directory by calling the `Ice.getSliceDir` function:

Python

```
Ice.loadSlice("-I" + Ice.getSliceDir() + " Color.ice")
```

This function attempts to locate the `slice` subdirectory of your Ice installation using an algorithm that succeeds for the following scenarios:

- Installation of a binary Ice archive
- Installation of an Ice source distribution using `make install`
- Installation via a Windows installer
- RPM installation on Linux
- Execution inside a compiled Ice source distribution

If the `slice` subdirectory can be found, `getSliceDir` returns its absolute path name, otherwise the function returns `None`.

Loading Multiple Slice Files in Python

You can specify as many Slice files as necessary in a single invocation of `Ice.loadSlice`, as shown below:

Python

```
Ice.loadSlice("Syscall.ice Process.ice")
```

Alternatively, you can call `Ice.loadSlice` several times:

Python

```
Ice.loadSlice("Syscall.ice")
Ice.loadSlice("Process.ice")
```

If a Slice file includes another file, the default behavior of `Ice.loadSlice` generates Python code only for the named file. For example, suppose `Syscall.ice` includes `Process.ice` as follows:

Slice

```
// Syscall.ice
#include <Process.ice>
...
```

If you call `Ice.loadSlice("-I. Syscall.ice")`, Python code is not generated for the Slice definitions in `Process.ice` or for any definitions that may be included by `Process.ice`. If you also need code to be generated for included files, one solution is to load them individually in subsequent calls to `Ice.loadSlice`. However, it is much simpler, not to mention more efficient, to use the `--all` option instead:

Python

```
Ice.loadSlice("--all -I. Syscall.ice")
```

When you specify `--all`, `Ice.loadSlice` generates Python code for all Slice definitions included directly or indirectly from the named Slice files.

There is no harm in loading a Slice file multiple times, aside from the additional overhead associated with code generation. For example, this situation could arise when you need to load multiple top-level Slice files that happen to include a common subset of nested files. Suppose that we need to load both `Syscall.ice` and `Kernel.ice`, both of which include `Process.ice`. The simplest way to load both files is with a single call to `Ice.loadSlice`:

Python

```
Ice.loadSlice("--all -I. Syscall.ice Kernel.ice")
```

Although this invocation causes the Ice extension to generate code twice for `Process.ice`, the generated code is structured so that the interpreter ignores duplicate definitions. We could have avoided generating unnecessary code with the following sequence of steps:

Python

```
Ice.loadSlice("--all -I. Syscall.ice")
Ice.loadSlice("-I. Kernel.ice")
```

In more complex cases, however, it can be difficult or impossible to completely avoid this situation, and the overhead of code generation is usually not significant enough to justify such an effort.

Static Code Generation in Python

You should be familiar with static code generation if you have used other Slice language mappings, such as C++ or Java. Using static code generation, the Slice compiler `slice2py` generates Python code from your Slice definitions.

Compiler Output in Python

For each Slice file `X.ice`, `slice2py` generates Python code into a file named `X_ice.py` in the output directory.

> Using the file name `X.py` would create problems if `X.ice` defined a module named `X`, therefore the suffix `_ice` is appended to the name of the generated file.

The default output directory is the current working directory, but a different directory can be specified using the `--output-dir` option.

In addition to the generated file, `slice2py` creates a Python package for each Slice module it encounters. A Python package is nothing more than a subdirectory that contains a file with a special name (`__init__.py`). This file is executed automatically by Python when a program first imports the package. It is created by `slice2py` and must not be edited manually. Inside the file is Python code to import the generated files that contain definitions in the Slice module of interest.

For example, the Slice files `Process.ice` and `Syscall.ice` both define types in the Slice module `OS`. First we present `Process.ice`:

Slice

```
module OS {
    interface Process {
        void kill();
    };
};
```

And here is `Syscall.ice`:

Slice

```
#include <Process.ice>
module OS {
    interface Syscall {
        Process getProcess(int pid);
    };
};
```

Next, we translate these files using the Slice compiler:

```
> slice2py -I. Process.ice Syscall.ice
```

If we list the contents of the output directory, we see the following entries:

Python

```
OS/
Process_ice.py
Syscall_ice.py
```

The subdirectory `OS` is the Python package that `slice2py` created for the Slice module `OS`. Inside this directory is the special file `__init__.py` that contains the following statements:

Python

```
import Process_ice
import Syscall_ice
```

Now when a Python program executes `import OS`, the two files `Process_ice.py` and `Syscall_ice.py` are implicitly imported.

Subsequent invocations of `slice2py` for Slice files that also contain definitions in the `OS` module result in additional `import` statements being added to `OS/__init__.py`. Be aware, however, that `import` statements may persist in `__init__.py` files after a Slice file is renamed or becomes obsolete. This situation may manifest itself as a run-time error if the interpreter can no longer locate the generated file while attempting to import the package. It may also cause more subtle problems, if an obsolete generated file is still present and being loaded unintentionally. In general, it is advisable to remove the package directory and regenerate it whenever the set of Slice files changes.

A Python program may also import a generated file explicitly, using a statement such as `import Process_ice`. Typically, however, it is more convenient to import the Python module once, rather than importing potentially several individual files that comprise the module, especially when you consider that the program must still import the module explicitly in order to make its definitions available. For example, it is much simpler to state

Python

```
import OS
```

rather than the following alternative:

Python

```
import Process_ice
import Syscall_ice
import OS
```

Include Files in Python

It is important to understand how `slice2py` handles include files. In the absence of the `--all` option, the compiler does not generate Python code for Slice definitions in included files. Rather, the compiler translates Slice `#include` statements into Python `import` statements in the following manner:

1. Determine the full pathname of the include file.
2. Create the shortest possible relative pathname for the include file by iterating over each of the include directories (specified using the `-I` option) and removing the leading directory from the include file if possible.
   For example, if the full pathname of an include file is `/opt/App/slice/OS/Process.ice`, and we specified the options `-I/opt/App` and `-I/opt/App/slice`, then the shortest relative pathname is `OS/Process.ice` after removing `/opt/App/slice`.
3. Replace any slashes with underscores, remove the `.ice` extension, and append `_ice`. Continuing our example from the previous step, the translated import statement becomes
   `import OS_Process_ice`

There is a potential problem here that must be addressed. The generated `import` statement shown above expects to find the file `OS_Process_ice.py` somewhere in Python's search path. However, `slice2py` uses a different default name, `Process_ice.py`, when it compiles `Process.ice`.. To resolve this issue, we must use the `--prefix` option when compiling `Process.ice`:

```
> slice2py --prefix OS_ Process.ice
```

The `--prefix` option causes the compiler to prepend the specified prefix to the name of each generated file. When executed, the above command creates the desired file name: `OS_Process_ice.py`.

It should be apparent by now that generating Python code for a complex Ice application requires a bit of planning. In particular, it is imperative that you be consistent in your use of `#include` statements, include directories, and `--prefix` options to ensure that the correct file names are used at all times.

Of course, these precautionary steps are only necessary when you are compiling Slice files individually. An alternative is to use the `--all` option and generate Python code for all of your Slice definitions into one Python source file. If you do not have a suitable Slice file that includes all necessary Slice definitions, you could write a "master" Slice file specifically for this purpose.

Static Versus Dynamic Code Generation in Python

There are several issues to consider when evaluating your requirements for code generation.

Application Considerations for Code Generation in Python

The requirements of your application generally dictate whether you should use dynamic or static code generation. Dynamic code generation is convenient for a number of reasons:

- it avoids the intermediate compilation step required by static code generation
- it makes the application more compact because the application requires only the Slice files, not the assortment of files and directories produced by static code generation
- it reduces complexity, which is especially helpful during testing, or when writing short or transient programs.

Static code generation, on the other hand, is appropriate in many situations:

- when an application uses a large number of Slice definitions and the startup delay must be minimized
- when it is not feasible to deploy Slice files with the application
- when a number of applications share the same Slice files
- when Python code is required in order to utilize third-party Python tools.

Mixing Static and Dynamic Code Generation in Python

Using a combination of static and dynamic translation in an application can produce unexpected results. For example, consider a situation where a dynamically-translated Slice file includes another Slice file that was statically translated:

Slice

```
// Slice
#include <Glacier2/Session.ice>

module App {
    interface SessionFactory {
        Glacier2::Session* createSession();
    };
};
```

The Slice file `Session.ice` is statically translated, as are all of the Slice files included with the Ice run time.

Assuming the above definitions are saved in `App.ice`, let's execute a simple Python script:

Python

```
# Python
import Ice
Ice.loadSlice("-I/opt/Ice/slice App.ice")

import Glacier2
class MyVerifier(Glacier2.PermissionsVerifier): # Error
    def checkPermissions(self, userId, password):
        return (True, "")
```

The code looks reasonable, but running it produces the following error:

```
'module' object has no attribute 'PermissionsVerifier'
```

Normally, importing the Glacier2 module as we have done here would load all of the Python code generated for the Glacier2 Slice files. However, since `App.ice` has already included a subset of the Glacier2 definitions, the Python interpreter ignores any subsequent requests to import the entire module, and therefore the `PermissionsVerifier` type is not present.

One way to address this problem is to import the statically-translated modules first, prior to loading Slice files dynamically:

Python

```
# Python
import Ice, Glacier2 # Import Glacier2 before App.ice is loaded
Ice.loadSlice("-I/opt/Ice/slice App.ice")

class MyVerifier(Glacier2.PermissionsVerifier): # OK
    def checkPermissions(self, userId, password):
        return (True, "")
```

The disadvantage of this approach in a non-trivial application is that it breaks encapsulation, forcing one Python module to know what other modules are doing. For example, suppose we place our `PermissionsVerifier` implementation in a module named `verifier.py`:

Python

```
# Python
import Glacier2
class MyVerifier(Glacier2.PermissionsVerifier):
    def checkPermissions(self, userId, password):
        return (True, "")
```

Now that the use of Glacier2 definitions is encapsulated in `verifier.py`, we would like to remove references to Glacier2 from the main script:

Python

```
# Python
import Ice
Ice.loadSlice("-I/opt/Ice/slice App.ice")
...
import verifier # Error
v = verifier.MyVerifier()
```

Unfortunately, executing this script produces the same error as before. To fix it, we have to break the `verifier` module's encapsulation and import the `Glacier2` module in the main script because we know that the `verifier` module requires it:

Python

```
# Python
import Ice, Glacier2
Ice.loadSlice("-I/opt/Ice/slice App.ice")
...
import verifier # OK
v = verifier.MyVerifier()
```

Although breaking encapsulation in this way might offend our sense of good design, it is a relatively minor issue.

Another solution is to import the necessary submodules explicitly. We can safely remove the Glacier2 reference from our main script after rewriting `verifier.py` as shown below:

Python

```
# Python
import Glacier2_PermissionsVerifier_ice
import Glacier2
class MyVerifier(Glacier2.PermissionsVerifier):
    def checkPermissions(self, userId, password):
        return (True, "")
```

Using the rules defined for static code generation, we can derive the name of the module containing the code generated for `PermissionsVerifier.ice` and import it directly. We need a second `import` statement to make the Glacier2 definitions accessible in this module.

`slice2py` Command-Line Options

The Slice-to-Python compiler, `slice2py`, offers the following command-line options in addition to the standard options:

- `--all`
  Generate code for all Slice definitions, including those from included files.
- `--checksum`
  Generate checksums for Slice definitions.
- `--prefix PREFIX`
  Use *`PREFIX`* as the prefix for generated file names.

Generating Packages in Python

By default, the scope of a Slice definition determines the module of its mapped Python construct. There are times, however, when applications require greater control over the packaging of generated Python code. For example, consider the following Slice definitions:

Slice

```
module sys {
    interface Process {
        // ...
    };
};
```

Other language mappings can use these Slice definitions as shown, but they present a problem for the Python mapping: the top-level Slice module `sys` conflicts with Python's predefined module sys. A Python application executing the statement `import sys` would import whichever module the interpreter happens to locate first in its search path.

A workaround for this problem is to modify the Slice definitions so that the top-level module no longer conflicts with a predefined Python module, but that may not be feasible in certain situations. For example, the application may already be deployed using other language mappings, in which case the impact of modifying the Slice definitions could represent an unacceptable expense.

The Python mapping could have addressed this issue by considering the names of predefined modules to be reserved, in which case the Slice module `sys` would be mapped to the Python module `_sys`. However, the likelihood of a name conflict is relatively low to justify such a solution, therefore the mapping supports a different approach: global metadata can be used to enclose generated code in a Python package. Our modified Slice definitions demonstrate this feature:

Slice

```
[["python:package:zeroc"]]
module sys {
    interface Process {
        // ...
    };
};
```

The global metadata directive `python:package:zeroc` causes the mapping to generate all of the code resulting from definitions in this Slice file into the Python package `zeroc`. The net effect is the same as if we had enclosed our Slice definitions in the module `zeroc`: the Slice module `sys` is mapped to the Python module `zeroc.sys`. However, by using metadata we have not affected the semantics of the Slice definitions, nor have we affected other language mappings.

See Also

- Using the Slice Compilers
- Python Mapping for Modules
- Using Slice Checksums in Python
- Metadata