

Writing an Ice Application with Python

This page shows how to create an Ice application with Python.

On this page:

- [Compiling a Slice Definition for Python](#)
- [Writing a Server in Python](#)
- [Writing a Client in Python](#)
- [Running Client and Server in Python](#)

Compiling a Slice Definition for Python

The first step in creating our Python application is to compile our [Slice definition](#) to generate Python proxies and skeletons. You can compile the definition as follows:

```
slice2py Printer.ice
```

The `slice2py` compiler produces a single source file, `Printer_ice.py`, from this definition. The compiler also creates a Python package for the `Demo` module, resulting in a subdirectory named `Demo`. The exact contents of the source file do not concern us for now — it contains the generated code that corresponds to the `Printer` interface we defined in `Printer.ice`.

Writing a Server in Python

To implement our `Printer` interface, we must create a servant class. By convention, a servant class uses the name of its interface with an `I`-suffix, so our servant class is called `PrinterI`:

Python

```
class PrinterI(Demo.Printer):
    def printString(self, s, current=None):
        print s
```

The `PrinterI` class inherits from a base class called `Demo.Printer`, which is generated by the `slice2py` compiler. The base class is abstract and contains a `printString` method that accepts a string for the printer to print and a parameter of type `Ice.Current`. (For now we will ignore the `Ice.Current` parameter.) Our implementation of the `printString` method simply writes its argument to the terminal.

The remainder of the server code, in `server.py`, follows our servant class and is shown in full here:

Python

```
import sys, Ice
import Demo

class PrinterI(Demo.Printer):
    def printString(self, s, current=None):
        print s

with Ice.initialize(sys.argv) as communicator:
    adapter = communicator.createObjectAdapterWithEndpoints("SimplePrinterAdapter", "default -p 10000")
    object = PrinterI()
    adapter.add(object, communicator.stringToIdentity("SimplePrinter"))
    adapter.activate()
    communicator.waitForShutdown()
```

The body of the main program contains a `with` block in which we place all the server code. If the code throws an exception, it will be handled by the Python interpreter which typically prints out the exception and then returns failure to the operating system.

The `Ice.Communicator` object implements the Python context manager protocol, which allows us to use the `with` statement for the initialization of the `Ice.Communicator` object. This ensures the `communicator.destroy` method is called when the `with` block goes out of scope. Doing this is essential in order to correctly finalize the Ice run time



Failure to call `destroy` on the communicator before the program exits results in undefined behavior.

The server code goes through the following steps:

1. We initialize the Ice run time by calling `Ice.initialize`. (We pass `sys.argv` to this call because the server may have command-line arguments that are of interest to the run time; for this example, the server does not require any command-line arguments.) The call to `initialize` returns an `Ice.Communicator` reference, which is the main object in the Ice run time.
2. We create an object adapter by calling `createObjectAdapterWithEndpoints` on the `Communicator` instance. The arguments we pass are `"SimplePrinterAdapter"` (which is the name of the adapter) and `"default -p 10000"`, which instructs the adapter to listen for incoming requests using the default transport protocol (TCP/IP) at port number 10000.
3. At this point, the server-side run time is initialized and we create a servant for our `Printer` interface by instantiating a `PrinterI` object.
4. We inform the object adapter of the presence of a new servant by calling `add` on the adapter; the arguments to `add` are the servant we have just instantiated, plus an identifier. In this case, the string `"SimplePrinter"` is the name of the Ice object. (If we had multiple printers, each would have a different name or, more correctly, a different *object identity*.)
5. Next, we activate the adapter by calling its `activate` method. (The adapter is initially created in a holding state; this is useful if we have many servants that share the same adapter and do not want requests to be processed until after all the servants have been instantiated.)
6. Finally, we call `waitForShutdown`. This call suspends the calling thread until the server is shut down. (For now, we will simply interrupt the server on the command line when we no longer need it, which terminates the server immediately.)

Writing a Client in Python

The client code, in `client.py`, looks very similar to the server. Here it is in full:

Python

```
import sys, Ice
import Demo

with Ice.initialize(sys.argv) as communicator:
    base = communicator.stringToProxy("SimplePrinter:default -p 10000")
    printer = Demo.PrinterPrx.checkedCast(base)
    if not printer:
        raise RuntimeError("Invalid proxy")

    printer.printString("Hello World!")
```

Note that the overall code layout is the same as for the server: we use the same `with` block. The code does the following:

1. As for the server, we initialize the Ice run time by calling `Ice.initialize`.
2. The next step is to obtain a proxy for the remote printer. We create a proxy by calling `stringToProxy` on the communicator, with the string `"SimplePrinter:default -p 10000"`. Note that the string contains the object identity and the port number that were used by the server. (Obviously, hard-coding object identities and port numbers into our applications is a bad idea, but it will do for now; we will see more architecturally sound ways of doing this when we discuss [IceGrid](#).)
3. The proxy returned by `stringToProxy` is of type `Ice.ObjectPrx`, which is at the root of the inheritance tree for interfaces. But to actually talk to our printer, we need a proxy for a `Demo::Printer` interface, not an `Object` interface. To do this, we need to do a down-cast by calling `Demo.PrinterPrx.checkedCast`. A checked cast sends a message to the server, effectively asking "is this a proxy for a `Demo::Printer` interface?" If so, the call returns a proxy of type `Demo.PrinterPrx`; otherwise, if the proxy denotes an interface of some other type, the call returns `None`.
4. We test that the down-cast succeeded and, if not, throw an error message that terminates the client.
5. We now have a live proxy in our address space and can call the `printString` method, passing it the time-honored `"Hello World!"` string. The server prints that string on its terminal.

Running Client and Server in Python

To run client and server, we first start the server in a separate window:

```
python server.py
```

At this point, we won't see anything because the server simply waits for a client to connect to it. We run the client in a different window:

```
python client.py
```

The client runs and exits without producing any output; however, in the server window, we see the "Hello World!" that is produced by the printer. To get rid of the server, we interrupt it on the command line for now.

If anything goes wrong, the client will print an error message. For example, if we run the client without having first started the server, we get something like the following:

```
Traceback (most recent call last):
  File "client.py", line 10, in ?
    printer = Demo.PrinterPrx.checkedCast(base)
  File "Printer_ice.py", line 43, in checkedCast
    return Demo.PrinterPrx.ice_checkedCast(proxy, '::Demo::Printer', facet)
ConnectionRefusedException: Ice.ConnectionRefusedException:
Connection refused
```

Note that, to successfully run the client and server, the Python interpreter must be able to locate the Ice extension for Python. See the Ice for Python installation instructions for more information.

See Also

- [Client-Side Slice-to-Python Mapping](#)
- [Server-Side Slice-to-Python Mapping](#)
- [The Current Object](#)
- [IceGrid](#)