

# Multi-Language Programming

## Part III - Combining Python with Flash Lite and Symbian C++

Felipe Andrade, Mikko Ohtamaa, Jussi Toivola and Mark Wilcox

Published by the Symbian Developer Network

Version: 1.0 – December 2008

<b>1</b>	<b>INTRODUCTION.....</b>	<b>2</b>
<b>2</b>	<b>EXTENDING PYTHON WITH SYMBIAN C++ .....</b>	<b>2</b>
	2.1 MOTIVATION.....	3
	2.2 HYBRID LANGUAGE DEVELOPMENT MODEL .....	3
	2.3 EXTENDING PYTHON.....	3
	2.4 DEBUGGING EXTENSIONS .....	15
	2.5 DEPLOYMENT .....	15
	2.6 INTERFACE GENERATORS.....	15
<b>3</b>	<b>EXTENDING FLASH LITE WITH PYTHON FOR S60.....</b>	<b>17</b>
	3.1 FLASH LITE AND THE FLYER FRAMEWORK .....	18
	3.2 WHY PYTHON FOR S60 WITH FLASH LITE? .....	19
	3.3 HOW IT WORKS .....	19
	3.4 WHEN TO USE THE FLYER FRAMEWORK .....	23
	3.5 PACKAGING AND SIGNING YOUR CONTENT.....	24
<b>4</b>	<b>CONCLUSIONS.....</b>	<b>25</b>

# 1 Introduction

This paper is the third in a series examining the possibilities of building Symbian OS applications by mixing multiple languages and runtime environments. The first paper<sup>1</sup> presented an overview of the runtimes that are available and the combinations which make sense. The paper also detailed a general pattern for implementing multi-language applications. The second paper<sup>2</sup> described two example solutions which combine the strengths of Flash Lite for the UI with either Java ME or native Symbian C++ to access more features of the platform.

In this paper we focus on the popular Python programming language. On Symbian OS, Python is available in the form of PyS60 on S60 devices, which is open source and actively maintained by engineers at Nokia. There is also PyUIQ for UIQ 2.1 and UIQ 3 devices, although the project on SourceForge ([sourceforge.net/projects/pyuiq/](http://sourceforge.net/projects/pyuiq/)) is only in an 'alpha' state and hasn't been updated for more than a year. The Python interpreter does not currently come pre-installed on any Symbian devices and must either be packaged with the application or installed beforehand by the user. PyS60 is based on a fairly old version of the Python interpreter (2.2.2 from October 2002), but it comes with a range of Symbian and S60-specific modules which enable very rapid prototyping of mobile applications. There is also a very active, enthusiastic and supportive developer community ([discussion.forum.nokia.com/forum/forumdisplay.php?f=102](http://discussion.forum.nokia.com/forum/forumdisplay.php?f=102)). For anyone new to Python or PyS60 programming there is an excellent introductory guide published by Symbian Press, *Mobile Python: Rapid prototyping of applications on the mobile platform* (more information is available at [developer.symbian.com/main/documentation/books/books\\_files/python/index.jsp](http://developer.symbian.com/main/documentation/books/books_files/python/index.jsp)).

From the perspective of multi-language programming, Python is interesting because it has a built-in extension mechanism, which makes it easy to wrap native Symbian C++ code and use it like any other Python module. In Section 2 of this paper, Mikko Ohtamaa and Jussi Toivola of Red Innovation ([www.redinnovation.com/](http://www.redinnovation.com/)) explain how to achieve this in detail. Another great feature of Python is that it's very easy to learn, making it a good option for Flash Lite developers looking to extend the functionality available in the standard player. There are powerful and simple-to-use features for socket programming and string manipulation that allow the implementation of a local server to communicate with Flash Lite and interpret commands in a few tens of lines of code. Even this effort is unnecessary though, since Felipe Andrade has released the Flyer Framework under the open source Apache License.<sup>3</sup> In Section 3, he explains what Flyer is and how you can use it.

## 2 Extending Python with Symbian C++

When two programming languages or two environments are co-operating, some sort of bridge must be formed between them: operating systems and user land applications have system calls and virtual machines running bytecode have native extensions. Written in a programming language which directly compiles into native binary format, the extensions offer a standardized way to wrap low level machine calls and byte structures to high level programming language constructs. The most notable extension mechanism today is JNI (Java Native Interface), which binds a Java runtime to its underlying operating system. Similarly, Python has Python extension modules. In this section we explore the possibility of extending the Python runtime with custom Symbian C++ elements.

---

<sup>1</sup> [developer.symbian.com/main/downloads/papers/Multi\\_Language\\_Programming\\_Part1\\_Overview.pdf](http://developer.symbian.com/main/downloads/papers/Multi_Language_Programming_Part1_Overview.pdf).

<sup>2</sup> [developer.symbian.com/main/downloads/papers/MLP\\_Paper+2\\_v1.0.pdf](http://developer.symbian.com/main/downloads/papers/MLP_Paper+2_v1.0.pdf).

<sup>3</sup> There is also another commercial alternative to the Flyer Framework called SWF2Go ([www.swf2go.com/](http://www.swf2go.com/)).

## 2.1 Motivation

Native programming on Symbian OS uses Symbian C++, which is known for its special and sometimes difficult constructs that are intended to reduce the software footprint and make code more robust. However, what is effective in terms of computing resources is not always effective in terms of development effort; for example, developers are forced to implement things like memory clean-up manually, which are automatic in more application-oriented programming languages.

Luckily for developers, Symbian OS provides a range of runtime environment options including Java, Flash and Python. They open up mobile development possibilities without the requirement to master a very complex skillset, promising a productivity boost in mobile software development.

Which brings us to the issue: higher level programming languages do not currently have automatic access to all the Symbian API goodies. You are stuck with the subset of possibilities provided by your current virtual machine runtime, especially in the case of Java ME, which is intentionally restrictive in terms of extension and is known for its slow adoption of new APIs (via the JSR process).

Unlike Java or Flash Lite, Python is not bound by this limitation. Because the Python implementation is open source and the runtime can be bundled with the application, you can extend the runtime with new features not provided by the standard Python libraries. Also, Python applications are not sandboxed and have all the same privileges as native Symbian applications.

## 2.2 Hybrid Language Development Model

Since Python development is productive<sup>4</sup> with its higher language constructs, it is often useful to at least prototype an application in Python when time-to-release is critical. It is not suggested that you cannot create production quality applications in Python; in fact, as the Python for S60 mobile runtime has matured we have started to see more and more real Python applications, like Niime ([www.niime.com/](http://www.niime.com/)) and IYOUIT ([www.iyouit.eu/](http://www.iyouit.eu/)), instead of just technology demos. In the Maemo community, built around Nokia's Linux-based Internet tablets, Python development is more the rule than the exception: there we have applications like the Canola media center ([openbossa.indt.org/canola/](http://openbossa.indt.org/canola/)), which is developed by the OpenBossa lab at INdT (Instituto Nokia de Tecnologia) in Brazil.

In a hybrid language model, most of the application logic and often the user interface are written in Python – only computationally sensitive parts and special native code are put into an extension module. This way we can have the best of the both worlds: the efficiency of Python development and the power of Symbian C++. The price tag is that if the native extension you need doesn't exist already, you need to know a Symbian C++ developer who can cook it up for you.

## 2.3 Extending Python

In the following sections we refer to Nokia's Python for S60 implementation, PyS60, when we talk about Python. We build an extension for wrapping a progress note dialog as an example.

### 2.3.1 Python application and extension structure

A SIS distributable Python application consists of a few standard parts. In order for the Python script to appear on the phone's menu, it needs a launcher. The launcher is a stub EXE which loads the Python runtime and bootstraps the Python code entry point. The Python runtime itself is

---

<sup>4</sup> Python won productivity awards in the *Software Development Jolt and Productivity Awards* in both 2000 and 2005.

distributed as a separate DLL. Native extension modules are also DLL files which have a function structure defined by the particular version of the Python virtual machine (compatibility is not guaranteed between major versions).

Our goal here is to build a new native extension which presents functions that can be called from Python and internally calls native Symbian methods. As part of the call, Python variables need to be read by native code and return values from native code back to Python. Creating the basic Python extension constructs does not differ much from normal extensions made for Linux or Windows, however some Symbian-specific constructs require special attention, such as the 'leave' mechanism. If you do not have any experience with Python C extensions, you should check the tutorial from the Python documentation at [www.python.org/doc/2.2/ext/intro.html](http://www.python.org/doc/2.2/ext/intro.html).

### 2.3.2 Progress notes dialog

Progress notes show different progress bars in S60 dialogs. The wrapped class is based on the example in the Forum Nokia wiki, at [wiki.forum.nokia.com/index.php/Progress\\_and\\_wait\\_notes](http://wiki.forum.nokia.com/index.php/Progress_and_wait_notes). Special thanks go to the contributors to that article.

For the sake of readability, we won't put the full source code of the example here. It can be found at [code.google.com/p/uikludges/source/browse/#svn/trunk/modules/progressnotes](http://code.google.com/p/uikludges/source/browse/#svn/trunk/modules/progressnotes).

### 2.3.3 Architecture

It is desirable to separate code that does useful work from code that simply interfaces with Python. This makes the code reusable for other Symbian C++ projects or indeed other runtimes with native interfaces such as Ruby.

In this example, we present a simple native process dialog to the user which can be displayed, updated and dismissed via a Python API.

The user interaction is handled by a class implementing the `MProgressDialogCallback` interface. In the Nokia wiki example, the interface was implemented by the same class, which created the dialogs. To keep the original code clean from Python references, we move the callback code to another class, which forwards the callback to the Python code.

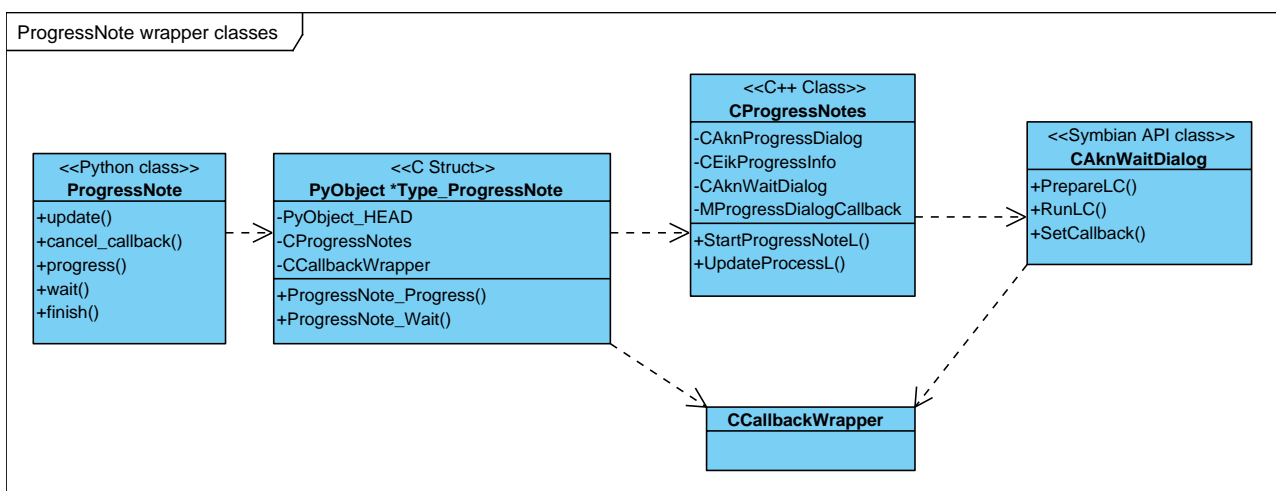


Figure 1: Wrapper class structure for progress notes showing some of the methods and attributes

The Python extension is split into two files: `progressnote.(cpp,h,rs)` and `progressnotemodule.cpp`. The original implementation is in `progressnote.cpp` and Python-related code is in `progressnotemodule.cpp`.

We'll explain the module structure from the bottom up.

### 2.3.4 C++ class to Python class

To wrap a C++ class you must create a Python type struct for it. Each Python type struct starts with the `PyObject_HEAD` macro, which adds default Python values for the struct. After that we can add our own variables. In this case, we need the wrapped Symbian C++ class object and our callback wrapper object associated with it.

```

/// CProgressNote class wrapper declaration
struct Type_ProgressNote
{
    /// Python type header.
    PyObject_HEAD;
    /// CProgressNotes instance.
    CProgressNotes* iProgressNotes;
    /// Our callback interface
    CCallbackWrapper* iCallbackWrapper;
};

```

### 2.3.5 C++ class methods to Python class methods

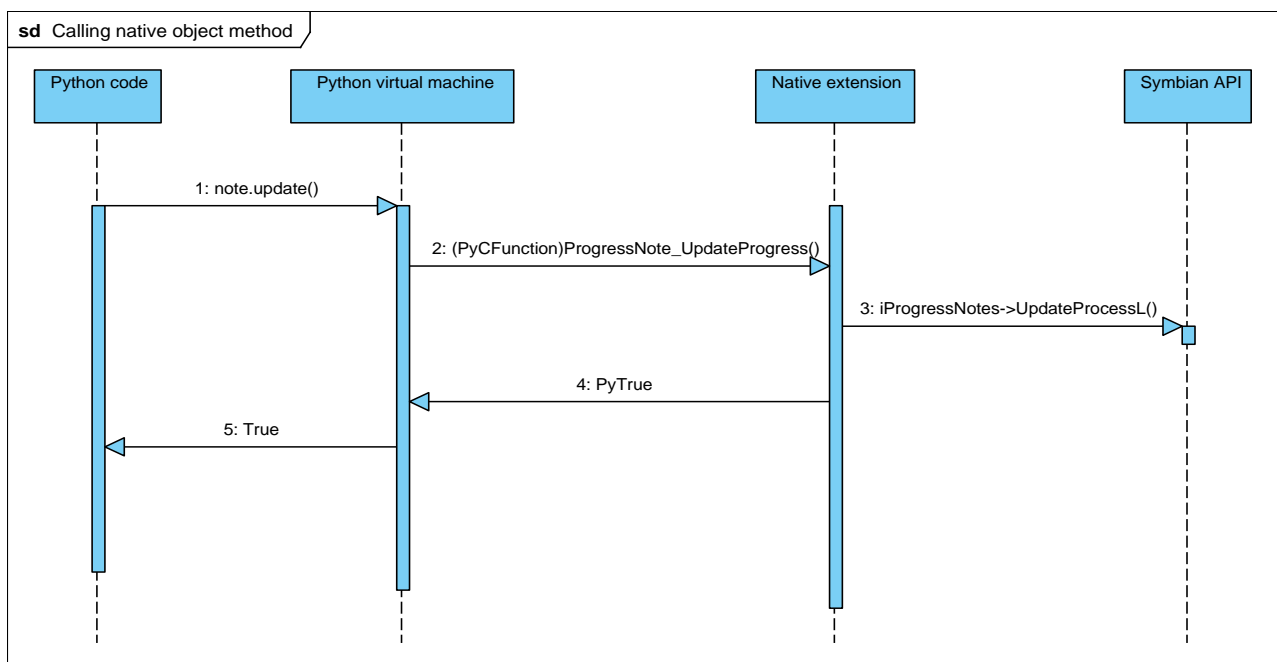


Figure 2: Native method call chain

Next we need to know which methods of the Symbian C++ class we want to access. For each method, we create corresponding C functions, which map Python methods to Symbian C++ methods. The functions are mapped in a `ProgressNote_methods` table.

The data fields are:

<b>Python method name</b>	Name of the method accessible from Python code.
<b>C function</b>	The respective native function.
<b>Parameter flag</b>	Parameter flags can tell what kind of arguments the native function accepts. There can be no arguments ( <code>METH_NOARGS</code> ), variable arguments ( <code>METH_VARARGS</code> ) and/or keyword arguments ( <code>METH_KEYWORDS</code> ). If you declare that a function takes arguments, but the C function is missing the required parameters, your application simply crashes when you try to call the function from Python code.
<b>Doc string</b>	Documentation

```
static const PyMethodDef ProgressNote_methods[] =
{
    {"progress", (PyCFunction)ProgressNote_Progress, METH_VARARGS, ""},
    {"update", (PyCFunction)ProgressNote_UpdateProgress, METH_VARARGS, ""},
    {"wait", (PyCFunction)ProgressNote_Wait, METH_NOARGS, ""},
    {"finish", (PyCFunction)ProgressNote_Finish, METH_NOARGS, ""},
    {"cancel_callback", (PyCFunction)ProgressNote_SetCancel_Callback,
        METH_VARARGS, ""},
    {NULL, NULL}
};
```

To access the methods, you must provide a `getattr()` method for your Python class. You must create a C function, which is registered for your Python class type. More about registering can be found in Section 2.3.8.

```
static PyObject *getattr_ProgressNote(PyObject* self, char* name)
{
    return Py_FindMethod(const_cast<PyMethodDef*>(&ProgressNote_methods[0]), self,
        name);
}
```

Next we'll need to implement the wrapper functions.

Here we wrap the `CProgressNotes::UpdateProcessL()` method, which updates the state of the progress note and changes its title. This function serves as a good example for Symbian C++ extension. We must read Python arguments, convert a C-style string into a Symbian C++ descriptor and catch Symbian C++ leaves, converting them into Python exceptions.

### 2.3.6 Catching Symbian leaves

If a wrapped Symbian function leaves, your Python code is not able to catch it. The `appui` fw module catches the leave eventually, but shows you a very uninformative message which is hard to debug if discovered and would produce a bad user experience otherwise. You must therefore remember to TRAP any leaving function and rethrow the Symbian C++ leave as a standard Python exception as follows:

```

SPyErr_SetFromSymbianOSErr().
static PyObject* ProgressNote_UpdateProgress(Type_ProgressNote* self,
                                             PyObject* args)
{
    TInt newpos      = 0;
    char* title      = NULL;
    TInt title_length = 0;

    if (!PyArg_ParseTuple(args, "iu#", &newpos, &title, &title_length))
    {
        return 0;
    }

    // Convert C string into Symbian descriptor
    TPtrC aTitle((TUInt16*)title, title_length);

    // TRAP leaves
    TRAPD(err, self->ProgressNotes->UpdateProcessL(newpos, aTitle));
    if (err != KErrNone)
    {
        // Returns NULL
        return SPyErr_SetFromSymbianOSErr(err);
    }
    // Remember to increase reference count
    Py_INCREF(Py_True);
    return Py_True;
}

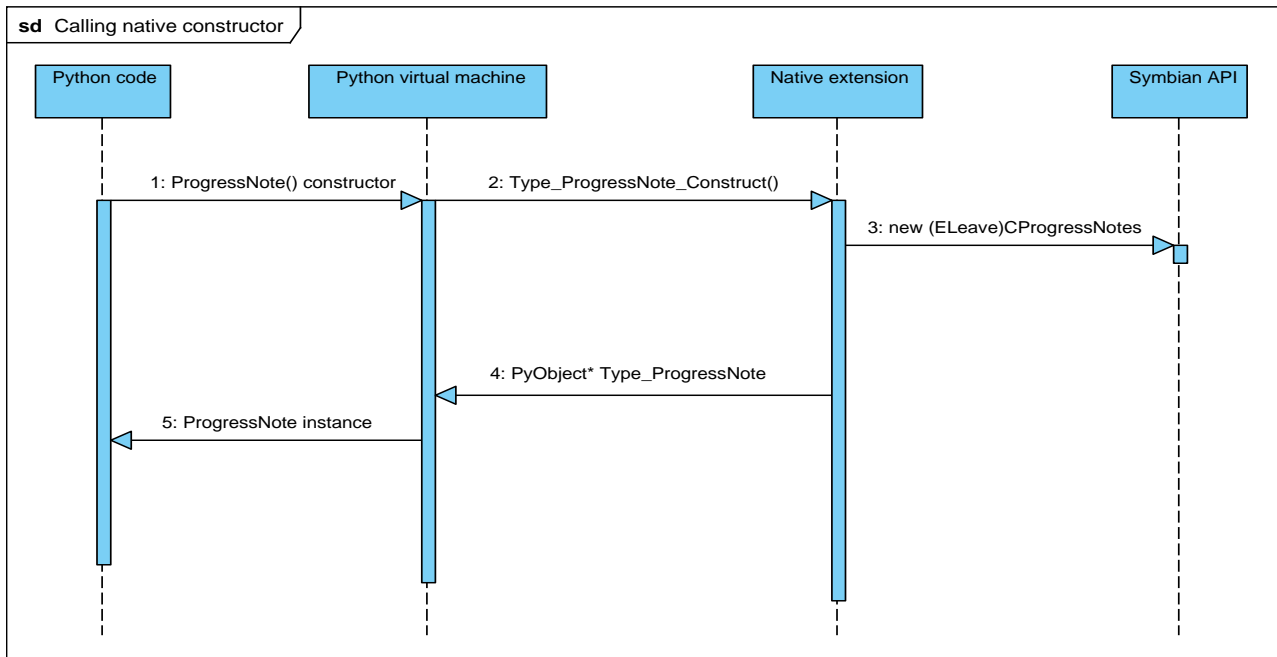
```

Note: in the function above the arguments are parsed with the format string 'iu#', which treats them as an integer (the 'i') followed by a UCS-2 string and its length (the 'u#'). When creating any strings for display in PyS60, using the built-in modules or your own extensions, you have to remember to use the prefix 'u' to denote Unicode, for example:

```
appui.fw.note(u"Success", "conf")
```

The second string, 'conf', doesn't require the prefix because it is used internally by the function called (in this case to specify the type of note) and is not displayed on the screen.

### 2.3.7 Python class constructor



**Figure 3: Creating native objects**

The constructor is a function, which returns an instance of the Python class.

```

/** ProgressNote python constructor */
static PyObject* Type_ProgressNote_Construct(PyObject* /*self*/, PyObject* args)
{
    Type_ProgressNote* self = PyObject_New(Type_ProgressNote,
                                             &progressnotes_ProgressNoteType);

    if (!self)
    {
        return SPyErr_SetFromSymbianOSErr(KErrNoMemory);
    }

    TRAPD(err, self->iProgressNotes = new (ELeave)CProgressNotes();
          self->iProgressNotes->ConstructL());
    if (err)
    {
        return SPyErr_SetFromSymbianOSErr(err);
    }
    // Class not derived from CBase, so not zero-initialized by default
    self->iCallbackWrapper = NULL;

    return (PyObject*)self;
}
  
```

### 2.3.8 Python class destructor

Here we destroy the wrapped Symbian C++ object and the respective callback wrapper.

```
/** ProgressNote python destructor */
static void dealloc_ProgressNote(Type_ProgressNote* self)
{
    if (self->iProgressNotes)
    {
        delete self->iProgressNotes;
        self->iProgressNotes = NULL;
    }
    if (self->iCallbackWrapper)
    {
        delete self->iCallbackWrapper;
        self->iCallbackWrapper = NULL;
    }
    PyObject_Del(self);
}
```

### 2.3.9 Create type for Python class

This is the place to register our 'getattr' and destructor functions to the Python class.

```
static const PyTypeObject progressnotes_ProgressNoteType =
{
    /******
    PyObject_HEAD_INIT(0)          /* initialize to 0 to ensure Win32 portability */
    0,                            /*ob_size, not used */
    "_progressnotes.ProgressNote", /*tp_name*/
    sizeof(Type_ProgressNote),    /*tp_basicsize*/
    0,                            /*tp_itemsize*/
    /* methods */
    (destructor)dealloc_ProgressNote, /*tp_dealloc*/
    0,                            /*tp_print*/
    (getattrfunc)getattr_ProgressNote, /*tp_getattr*/

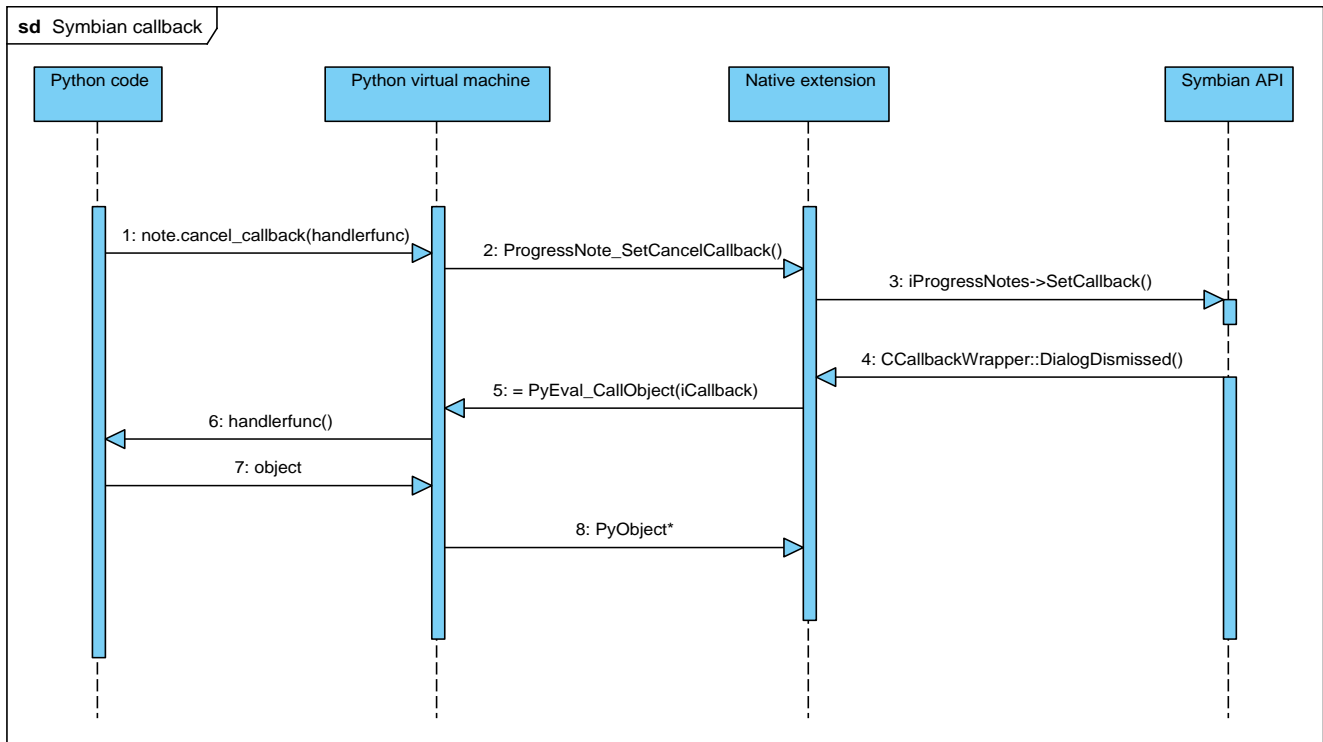
    /* implied by ISO C: all zeros thereafter */
};
```

The constructor is given in the module's function table, making the function importable from Python code.

```
static const PyMethodDef _progressnotes_methods[] =
{
    {"ProgressNote", (PyCFunction)Type_ProgressNote_Construct, METH_VARARGS},
    {NULL,          NULL}          /* sentinel */
};
```

### 2.3.10 Symbian C++ callback to Python callback

In order to detect if the user has canceled the operation, we need to send a signal to the Python code so that our application logic can handle the situation. For this, we use a Python callback function called from the Symbian C++ code. We create a wrapper for the Python class by using a Symbian C++ class, which implements the `MProgressDialogCallback` interface. Instead of defining application logic in that class, we simply call the Python callback function.



**Figure 4: Symbian C++ to Python callbacks**

```

void DialogDismissedL (TInt aButtonId)
{
    iParent->FinishL(EFalse);

    if (aButtonId == EAknSoftkeyCancel)
    {
        // See: http://www.python.org/doc/ext/callingPython.html
        PyObject* result;

        // Restore Python context before calling Python code!
        PyEval_RestoreThread(PYTHON_TLS->thread_state);
        result = PyEval_CallObject(iCallback, NULL);
        PyEval_SaveThread();

        // Reduce reference to Python object
        Py_XDECREF(result);
    }
}

~CCallbackWrapper()
{
    // Reduce reference to callback object
    Py_XDECREF(iCallback);
};

PyObject* ProgressNote_SetCancelCallback(Type_ProgressNote* self, PyObject* args)

```

```

{
    PyObject* callback = NULL;
    if (!PyArg_ParseTuple(args, "O", &callback))
    {
        return NULL;
    }

    self->callbackWrapper = new CCallbackWrapper(callback, self->ProgressNotes);
    if (!self->callbackWrapper)
    {
        return SPyErr_SetFromSymbianOSErr(KErrNoMemory);
    }

    self->ProgressNotes->SetCallback(self->callbackWrapper);

    Py_INCREF(Py_True);
    return Py_True;
}

```

### 2.3.10.1 Calling a callback

Before calling our callback we must remember to restore the Python context. Without doing so, the application crashes with a Kern-Exec 3 panic whenever the user presses 'Cancel.' Our callback takes no parameters so we supply NULL instead. To send parameters, you'll need to construct a Python tuple.

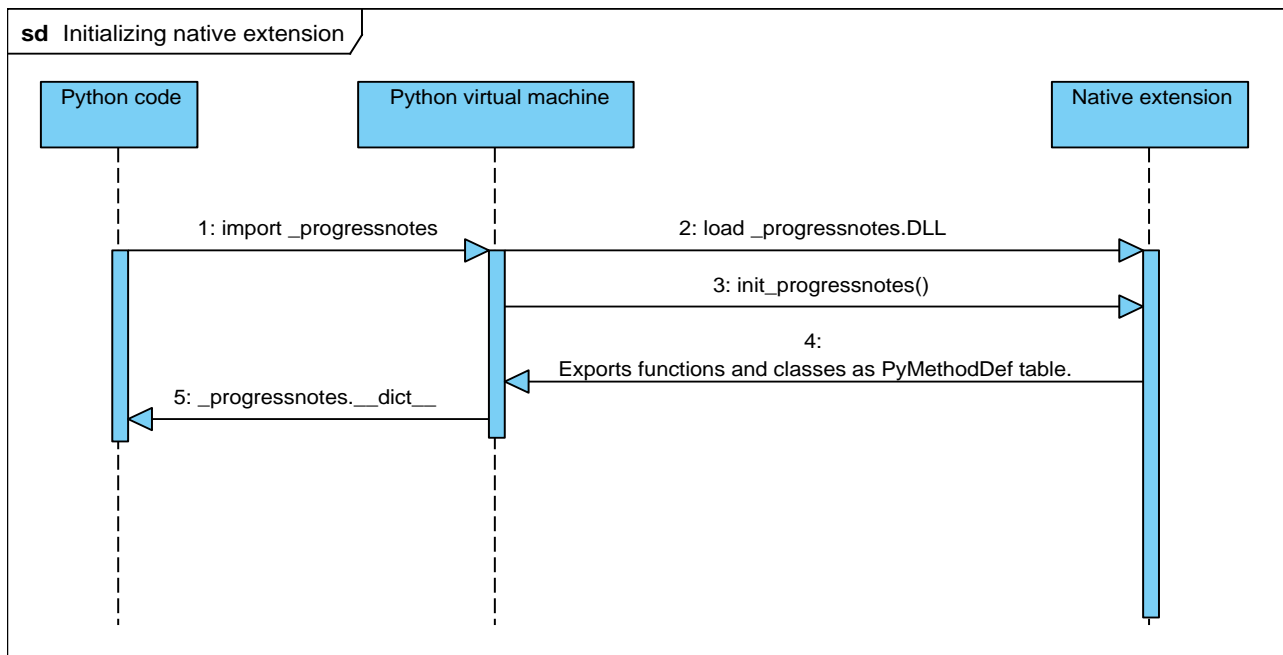
### 2.3.10.2 Handling the callback result

Python uses reference counting-based garbage collection. When we deal with Python objects in the native domain, we need to make sure that garbage collection is made aware of our operations by using the Py\_INCREF and Py\_DECREF macros.

Python has two, similar, operations for reducing the reference: Py\_XDECREF and Py\_DECREF. Py\_XDECREF is a simple macro, which checks if the object is NULL and if it is not, Py\_DECREF is called. If NULL is returned from a Python callback, it is a sign that a Python exception was raised. This is explained at [www.python.org/doc/2.2.2/ext/callingPython.html](http://www.python.org/doc/2.2.2/ext/callingPython.html).

In our C extension there is no need to check if the returned value is NULL. The exception can be caught in Python code where the script yields to the other active objects in the system. A call to e32.ao\_sleep() puts the current active object to sleep and allows other active objects to execute, including the S60 user interface, which handles the native Cancel callback and calls our Python callback. You can catch exceptions raised in the Python callback by setting try-except around the e32.ao\_sleep().

### 2.3.11 Initializing a module



**Figure 5: Initializing Python extension**

The following code is called when the DLL is dynamically loaded by Python.

```

DL_EXPORT(void) init_progressnotes(void)
{
    // &PyType_Type is not constant so we have to assign it here
    // instead of in the progressnotes_ProgressNoteType definition.
    progressnotes_ProgressNoteType.ob_type = &PyType_Type;

    // Initialize module
    PyObject* m = Py_InitModule("_progressnotes",
                               (PyMethodDef*)_progressnotes_methods);
}
  
```

Best practice is to create a wrapper in Python with the name 'progressnotes' and reserve the underscored name '\_progressnotes' for the extension module. This makes it easier to design a good extendable API. In our case we just create a 'progressnotes.py' file, which goes in the Python library folder C:\resource\.

Since we do not add any documentation or copyright notices into the Python code, we simply export all extension module functions and classes from the native extension as is. Importing `_progressnotes` triggers the native DLL loading, matched by name, and our `init_progressnotes()` DL\_EXPORT function (again matched by name<sup>5</sup>) is called, which puts native symbols into the module look-up table.

<sup>5</sup> Currently the module initialization function `init_progressnotes()` must be at the first ordinal of the extension DLL, since Symbian OS only uses symbol look-up by ordinal, prior to the new target types in Symbian OS version 9.3.

The module initialization function `init_progressnotes()` must be at the first ordinal of the extension DLL. You can also define an optional finalizer function at the second ordinal. A finalizer is rarely needed, but it may be useful if you happen to have global variables in your DLL that must be destroyed, i.e.,

```
from _progressnotes import *
```

### 2.3.12 Running the module

Now we are ready to give the extension a spin. The following application displays a progress note with increasing value. If the user cancels, the callback is called to signal cancellation in the processing loop. The Cancel Status class is a callback function wrapper where we can put our status variables – this way we avoid using globals.

```
import e32
import progressnotes

note = progressnotes.ProgressNote()

class CancelStatus:

    def __init__(self):
        self.cancelled = False

    def __call__(self):
        self.cancelled = True

status = CancelStatus()

# Callback must be given before opening dialog.
note.cancel_callback(status)
# Maximum progress
maxval = 20
# Open the note
note.progress(maxval)
# Execute your code here and update the progress note.
for x in xrange(maxval):
    note.update(x + 1, u"Progress %d\\%d" % (x + 1, maxval))
    e32.ao_sleep(0.1)
    if status.cancelled: break
# Close the note
note.finish()
```




---

Figure 6: S60 progress note accessed from Python

---

### 2.3.13 Another example

Another native type of note is the wait note, which can be wrapped and accessed from Python in exactly the same way.

See [code.google.com/p/uikludges/source/browse/trunk/modules/progressnotes/test\\_progressnotes.py](http://code.google.com/p/uikludges/source/browse/trunk/modules/progressnotes/test_progressnotes.py) for the source code.




---

Figure 7: S60 wait note accessed from Python

---

## 2.4 Debugging Extensions

Debugging extensions is done in the same way as debugging any Symbian C++ code. The most basic way is to use file logging (the `RFileLogger` class). A tutorial is available at [www.newluc.com/Creating-log-files.html](http://www.newluc.com/Creating-log-files.html). RDebug can be used on the emulator and is suitable for simple logging. It sends your logs to `%TEMP%\epocwind.out`, which can be read using any text editor. On a device, RDebug sends output to a serial port; this might be a bit troublesome to read for many developers.

Tools exist to send logging data in real time from device to the PC. One such tool is the open source LogMan utility ([code.google.com/p/logman-for-symbian/](http://code.google.com/p/logman-for-symbian/)), which can send logging data through a USB connection. It also provides a simple remote shell to manage the file system and processes.

## 2.5 Deployment

Deploying PyS60 applications can be difficult, especially when using custom extension modules. The default buildchain creates one SIS file per extension, leading to complex configuration management or a cumbersome end-user experience (as each SIS file needs to be installed by the user). Additional installation conflicts are raised by previous Python installations if the Python interpreter is bundled with the application.<sup>6</sup>

Python for S60 Community Edition ([launchpad.net/pys60community](http://launchpad.net/pys60community)) is an enhanced fork of Python for S60 by Nokia with an Scons-based build system ([code.google.com/p/scons-for-symbian/](http://code.google.com/p/scons-for-symbian/)) that is suitable for production deployment. Instead of deploying the Python runtime and extensions as separate SIS files, PyS60 Community Edition ([blog.redinnovation.com/2008/09/01/introducing-python-for-series-60-community-edition/](http://blog.redinnovation.com/2008/09/01/introducing-python-for-series-60-community-edition/)) builds one monolithic executable and SIS file for the application containing a runtime and extensions, which does not conflict with the standard Python for S60 installation.

## 2.6 Interface Generators

The majority of the work required to bind these two worlds together goes into mapping native data types to Python objects. Since in 90% of cases there exists a direct relationship between objects – say, Python unicode string to Symbian `TPtrC16` data type – it could be very useful to automate the mappings. This is where interface generators step in. They are tools that produce the extension code based on static analysis and in-comment tags.

To get an idea as to why this approach, using tools such as SWIG and SIP, is important, you just have to look at the sizes of the example files. The Python module is roughly three times larger than the wrapped Symbian C++ class.

---

<sup>6</sup> This is caused by the form of the official releases of the interpreter SIS files.

Interface generator	Pros	Cons
Manual wrapping and typing bridge by hand.	Total control allows workarounds for corner cases.	Creates lots of fragile bridge code with maintenance overhead.
Cog - ad hoc template language <a href="http://nedbatchelder.com/code/cog/">nedbatchelder.com/code/cog/</a>	Can be used to generate default interface code and add customizations in-line (in source comments). For example, see LogMan project.	Not a real interface generator.
SWIG <a href="http://www.swig.org/">www.swig.org/</a>	Simple Wrapper and Interface Generator. Very mature and has out-of-the-box Python support.	Does not play well with Unicode. Mapping between Python strings and Symbian descriptors is difficult. In our test, generated memory leaking code.
SIP <a href="http://www.riverbankcomputing.co.uk/software/sip/intro">www.riverbankcomputing.co.uk/software/sip/intro</a>	Very powerful, used in PyQt (for desktop systems).	Requires Python 2.3 (PyS60 is currently Python 2.2).
Boost.Python and Py++ <a href="http://www.language-binding.net/pyplusplus/pyplusplus.html">www.language-binding.net/pyplusplus/pyplusplus.html</a>	Very little manual work.	Porting the Boost Python Library to Symbian using Open C/C++ is a prerequisite.

Interface generators, combined with P.I.P.S.<sup>7</sup> or Open C/C++,<sup>8</sup> offer the possibility of adding lots of useful extensions based on open source libraries. Using these interface generators, or porting existing Python bindings for more capable UI toolkits such as PyQt4<sup>9</sup> (when Qt is available on Symbian OS) or pygame<sup>10</sup> (which depends on the Simple DirectMedia Layer library that's already ported<sup>11</sup>), it will also be fairly easy to improve the UI features of PyS60. However, currently there is only official support for a limited subset of the S60 UI components and a basic canvas. As an alternative, the designer-focused UI development tools and dynamic user interface features of Flash Lite provide an attractive option to combine with Python. Similarly, the modular nature and extensibility of Python make it attractive for Flash Lite developers looking for ways to escape their sandbox.

<sup>7</sup> [developer.symbian.com/wiki/display/pub/P.I.P.S.](http://developer.symbian.com/wiki/display/pub/P.I.P.S.)

<sup>8</sup> [www.forum.nokia.com/Resources\\_and\\_Information/Explore/Runtime\\_Platforms/Open\\_C\\_and\\_C++/](http://www.forum.nokia.com/Resources_and_Information/Explore/Runtime_Platforms/Open_C_and_C++/)

<sup>9</sup> [wiki.python.org/moin/PyQt4](http://wiki.python.org/moin/PyQt4)

<sup>10</sup> [www.pygame.org/news.html](http://www.pygame.org/news.html)

<sup>11</sup> [koti.mbnet.fi/mertama/sdl.html](http://koti.mbnet.fi/mertama/sdl.html)

### 3 Extending Flash Lite with Python for S60

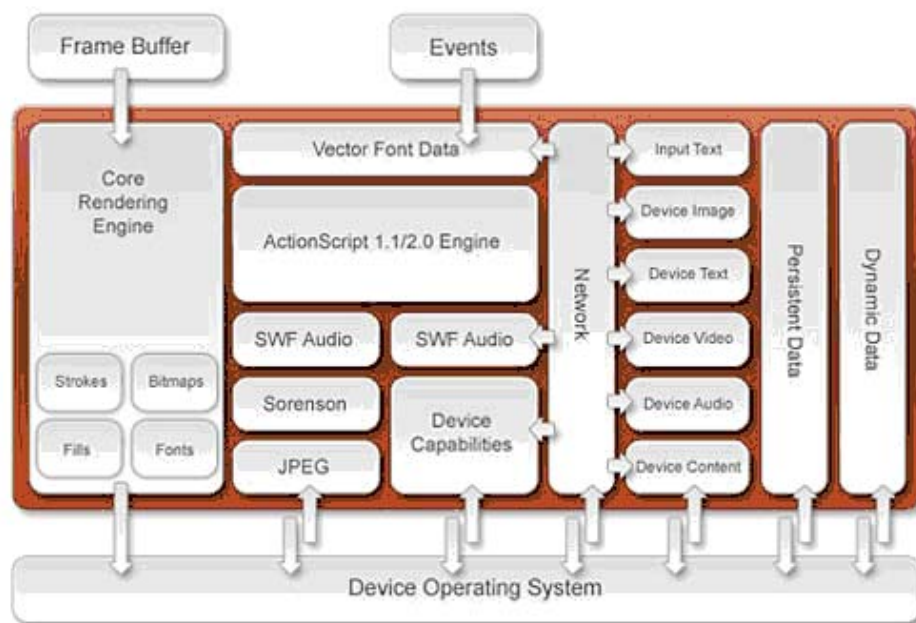
Combining Flash Lite and Python promises an extraordinary combination of clarity and productivity when joined with the power of open software. If you're willing to put in the necessary time and effort to understand how they can be bound together then it's possible to gain the advantages of both languages. One of the great aspects of Flash is the ability to work with scalable graphics that can be used to create smooth animations and custom user interfaces that run across many screen resolutions. This ability is perhaps the wow factor for Flash adoption in mobile phones. The small and varied screen resolutions are one of the most obvious challenges found in mobile programming but you can use Flash to create very attractive user interfaces in a relatively short time compared with other programming environments.

The Flyer Framework was created with the intention of providing the attractive graphical interface of Flash with better access to platform services, in order to create richer user experiences.

Flyer is open source software that joins Flash and Python together on mobile phones, the web and desktop platforms. Combining Flash Lite rapid development with the Adobe Flash IDE and the powerful but simple Python programming language makes iterative software prototyping relatively easy.

The Flyer Framework was primarily built to extend Flash Lite features with Python for S60. However, over time the framework has been extended to work on more platforms, empowering web and mobile developers to leverage the same user experience across any device with a display that supports Flash and Python. In the scope of this paper we will only talk about the mobile version.

Unfortunately, the design of the Adobe Flash Lite runtime only allows Flash applications to interact with very few native platform features, as we can see in Figure 8 below.



**Figure 8: Flash Lite architecture**

Many Flash Lite developers have been using the platform to build applications that run in the standalone player, rather than building content for the browser plug-in. With their latest releases Adobe appear to be continuing their focus on web-based content and the mobile browser experience, leaving native platform extensions to the device manufacturers that license their player technology.

### 3.1 Flash Lite and the Flyer Framework

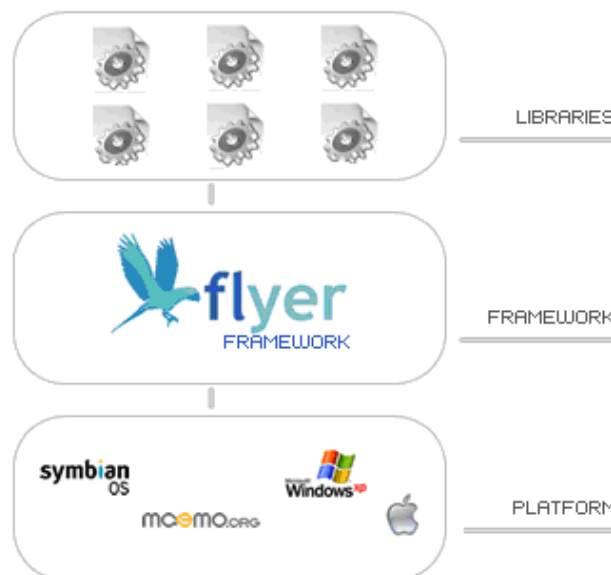
Flash Lite versions, development tools and extension mechanisms were covered in the second paper in this series, *Multi-Language Programming: Part II – Extending Flash Lite with Java ME or Symbian C++*. If you're not familiar with Flash Lite already then please refer to that paper.<sup>12</sup> The latest versions of Flash Lite have the following key features:

- FLV (Flash Video) support, including H.264, On2 VP6 and Sorenson video codecs
- enables a more complete web experience on mobile devices by providing access to content and video created with Adobe Flash
- faster performance
- object-based extension mechanism for accelerated UI design
- multiplatform support.

The S60 5th Edition SDK includes a Platform Services API that provides access to many more native platform features for Flash Lite developers on top of the standard player functionality. However, this API is specific to S60 5th Edition and is not available on earlier versions of the platform.

Developers looking to access more complex native features, like Bluetooth and the camera, from Flash Lite applications have already created powerful frameworks to enable mobile Flash developers to extend their applications in innovative ways. In the second paper of this series, referred to above, we discussed existing solutions that enable the creation of such applications without a single line of native code, such as KunerLite Rapid Application Development toolkit for Flash Lite. The Flyer Framework provides an open source alternative using Python, supporting several native functions out-of-the-box. The disadvantage of using Flyer is that pre-installation of the PyS60 runtime is required, but it also brings the advantage of being much easier to extend Flash Lite with Python, Standard C/C++ and native Symbian C++.

The following image explains a bit more about the Flyer Framework architecture.



**Figure 9: Flyer Framework architecture**

<sup>12</sup> [developer.symbian.com/main/downloads/papers/MLP\\_Paper+2\\_v1.0.pdf](http://developer.symbian.com/main/downloads/papers/MLP_Paper+2_v1.0.pdf).

The core Flyer plug-in runs on Windows, Linux, Mac OS X and Symbian OS and can be integrated with Adobe AIR, Adobe Flex, Adobe Flash Lite projects and many other environments, since it establishes a connection via TCP sockets.

### 3.2 Why Python for S60 with Flash Lite?

Python for S60 is, as the name suggests, limited to S60 devices, although the Python interpreter is already available on many other platforms. It's an open source and high level programming language designed to emphasize programmer productivity and code readability. The S60 port provides access to many of the platform's smartphone functions, such as camera, contacts, calendar, audio recording and playback, TCP/IP, Bluetooth communications and simple telephony.

On the other side, Flash Lite content can be developed in surprisingly little time with the Adobe Flash IDE,<sup>13</sup> vector graphics (which allows for scaling, rotation and other transformations without loss of graphic quality) and the ability to convert web-based Flash content to mobile and vice versa.

Flyer is flexible, extensible and free for commercial use and the current Apache license lets developers modify and extend the framework. It seems to be a reasonable choice to develop complex mobile applications with a short deadline and budget.

The current source code available is published under the Apache 2.0 license but it is just a piece of the framework for mobile devices; to get a free educational version of the full framework, [contact i2tecnologia](#). At the time of writing, the full version of Flyer is not available for commercial development.

### 3.3 How it Works

As we discussed in the second paper of this series, Flash Lite can be extended in various ways but providing a local TCP or HTTP server seems to be the most portable choice.

Before we get into details of packaging and signing, we will be introduced to the subject with an overview of the application model. It can be divided into two steps:

1. launch Flash content from Python for S60
2. make a bridge between Python for S60 and Flash Lite.

The following code snippet shows how to launch a Flash application from a Python for S60 application.

```
import appui fw, os
def close():
    appui fw. app. set_ exit ()
pathToFile = os. path. join(os. getcwd(), "main. swf")
appui fw. Content_ handl er(close). open(pathToFile)
```

We have to set up a socket in Python to create a bridge for Flash applications to request features not implemented natively on the Flash Lite side. This communication channel allows Flash Lite developers to offer more attractive applications by using sensors, Bluetooth or other native feature functionality, depending on the features implemented on the phone.

The open source version of Flyer sets up the communication channel when the Flyer class is created, as follows:

---

<sup>13</sup> Once you've got through the initially steep learning curve for the IDE.

```

import appui fw
import e32

class Flyer:
    def __init__(self):
        import socket
        serv = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        serv.bind(('127.0.0.1', 9100))
        serv.listen(999)
        self.serv = serv
        self.running = 0
        self.commands = {}
        self.state = None

```

Note the empty command array. When we start the server running, we set up the list of commands that we will accept and listen for an incoming connection.

```

def start(self):
    ...
    self.commands["sayText"] = self.sayText
    ...

    self.running = 1
    while (self.running == 1):
        print u"Flyer 1.0.35 - Waiting for FL connection"
        (client, addr) = self.serv.accept()
        self.client = client
        self.addr = addr
        print u"Connection opened with ", self.addr
        self.client.recv(1024, cb=self.handle)

```

Note that when we receive a message, we pass it to a function called `handle`, which parses the message, dispatches an appropriate command and then waits for another message, as follows:

```

def handle(self, msg=None):
    msg = msg.replace("\0", "")
    data = msg.split("|")

    if msg is not None:
        cmd = data[0] # command name
        filePath = data[1] # filePath or data string

    try:
        self.commands[cmd](filePath)
    except KeyError:
        print "Invalid Command"
    self.client.recv(1024, cb=self.handle)

```

If the message `sayText|hello` is passed to the server then it will call the function `sayText` with `hello` as an argument. Here's the definition of `sayText`:

```

def sayText(self, sentence):
    from flyersound import FlyerSound
    fSound = FlyerSound()
    fSound.say(sentence, callBackSound=self.broadcastMessage)
    self.state = fSound

```

This function uses one further function from the Flyer class, `broadcastMessage`, as the callback for a function in another class, `FlyerSound`, which is implemented in a separate module. Here's `broadcastMessage`:

```
def broadcastMessage(self, text):
    text += '\0'
    self.client.send(text)
```

It simply adds a NULL terminator to the string before sending it. The `FlyerSound` class handles all of the audio related commands for Flyer, what follows is just the part relating to `sayText` where you can see the callback above used.

```
from appuifw import *
from audio import *
import audio

class FlyerSound:
    ...
    def say(self, sentence=None, callbackSound=None):
        if sentence is None:
            sentence = query(u'Please provide the text', 'text')
        try:
            audio.say(sentence)
            if callbackSound is not None:
                callbackSound(u'Audio finished.')
        except Exception, error:
            if callbackSound is not None:
                callbackSound(unicode(error[1]))
            else:
                print unicode(error[1])
        ...
```

It's worth mentioning that the 'audio' and 'appuifw' modules imported here, as well as the 'e32' module imported by the main Flyer module, are included in the standard PyS60 distribution, so this really is everything required on the server side to implement the command handling. All that remains is to create and start the main Flyer class instance, like this:

```
flyer = Flyer()
flyer.start()
```

The matching code on the Flash Lite side of the connection is also fairly short and simple, making use of the `Delegate` and `EventDispatcher` classes provided by Adobe.

So, on the Flash side we set up an event dispatcher mechanism to enable Flash developers to select any combination of functionality depending on the features implemented by the framework.

```
import mx.utils.Delegate;
import mx.events.EventDispatcher;

class com.flyer.Flyer extends EventDispatcher {
    private static var SAY_SOUND: String = "sayText";
    ...

    public function Flyer(host_str: String, port_num: Number) {
        this.host_str = host_str;
    }
}
```

```

        this.port_num = port_num;

        this.flyerSocket_xmls = new XMLSocket();
    }

    public function connect():Void {
        this.flyerSocket_xmls.connect(this.host_str, this.port_num);
        this.flyerSocket_xmls.onConnect = Delegate.create(this, onConnect);
        this.flyerSocket_xmls.onClose = Delegate.create(this, onClose);
    }

    public function onConnect(success_bool:Boolean):Void {
        setStatus(true);
        dispatchEvent({type: "onFlyerConnect", status:success_bool});
    }

    public function onSayData(data_str:String):Void {
        dispatchEvent({type: "onSayData", data:data_str});
    }

    public function onClose():Void {
        setStatus(false);
        dispatchEvent({type: "onFlyerClose"})
    }

    public function sayText(sentence_str:String):Void {
        this.flyerSocket_xmls.send(Flyer.SAY_SOUND + "|" + sentence_str);
        this.flyerSocket_xmls.onData = Delegate.create(this, onSayData);
    }

    public function setStatus(status_bool:Boolean):Void {
        this.isConnected_bool = status_bool;
    }

    public function getStatus():Boolean {
        return this.isConnected_bool;
    }
}

```

To access data from the Python server you only have to import the Flyer ActionScript class and handle all events related to functionality used in the application.

```

import com.flyer.Flyer;

var speech_bool:Boolean = false;
var flyerConn_fl:Flyer = new Flyer('127.0.0.1', 9100);

var handler_obj:Object = new Object();
handler_obj.onFlyerConnect = function(evt_obj:Object):Void {
    preloader_mc._visible = false;
    if(evt_obj.status) {
        flyerConn_fl.sayText(textToSpeech_txt.text);
    } else {
        status_txt.text = "Connection failed.";
    }
}
}

```

```

handler_obj.onSayData = function(evt_obj: Object): Void {
    status_txt.text = evt_obj.data;
    speech_bool = false;
}

handler_obj.onFlyerClose = function(evt_obj: Object): Void {
    status_txt.text = "onFlyerClose";
}

flyerConn_fl.addEventLi stener("onFlyerConnect", handler_obj);
flyerConn_fl.addEventLi stener("onFlyerClose", handler_obj);
flyerConn_fl.addEventLi stener("onSayData", handler_obj);

flyerConn_fl.connect();

```

If you are planning to extend the Flyer Framework by providing your own Python extensions, be aware that your scripts will be exposed as plain text and so source code IP cannot be protected, since there are many tools available that are able to unpack the data and code from the SIS file. Since Python is an interpreted language, the simplest solutions are to have a business model that doesn't depend on the keeping your source code secret, or to use an obfuscator<sup>14</sup> and compile your Python scripts to bytecode. More information about this can be found at [wiki.forum.nokia.com/index.php/How\\_to\\_compile\\_PyS60\\_scripts\\_and\\_libraries\\_to\\_bytecode](http://wiki.forum.nokia.com/index.php/How_to_compile_PyS60_scripts_and_libraries_to_bytecode).

### 3.4 When to use the Flyer Framework

The Flyer Framework can be used to create highly engaging mobile content by using the Python application as a service provider and Flash Lite as UI presentation layer (full Flash UI and Python services running in background mode).

In addition to providing easy interaction between Flash and Python, Flyer has many special features to help you to find exactly what you're looking for:

- Bluetooth
- GPS
- device camera
- contacts
- calendar
- SMS
- your own library. Flyer is open source software and can be extended by providing Python libraries or native Symbian C++ libraries as described in Section 2 of this paper.

Flyer provides an integration of Flash Lite (1.1, 2.x, 3.0) with Python for S60. Some samples can be found at [code.google.com/p/flyer/](http://code.google.com/p/flyer/). The current source only works with Flash Lite 2.x or 3.x-enabled devices since it is using XMLSocket to exchange data. The full version (see Section 3.2) works on all Flash Lite versions.

---

<sup>14</sup> However, Python is not well suited for obfuscation, since any portion of any string can be interpreted as a variable or function name. This makes general Python obfuscation very difficult.

### 3.5 Packaging and Signing Your Content

The following image illustrates the key steps to get your application on to the market once you have got all of the code ready.



**Figure 10: Steps for packaging and signing**

#### 3.5.1 How to package Python and Flash Lite content for Symbian devices

To provide a simple installation method where a consumer would install a SIS file and not even know that the Flyer Framework was packaged within the installation, we have used the open source Ensymble developer utilities ([www.nbl.fi/~nbl928/ensymble.html](http://www.nbl.fi/~nbl928/ensymble.html)) to generate SIS packages and merge them with the Python runtime, which is available at Source Forge ([sourceforge.net/projects/pys60](http://sourceforge.net/projects/pys60)).

The following code snippet shows how to create a Python for S60 program using the Ensymble command line. In this case, all of the necessary files (Python scripts, compiled Flash Lite content and resources) must be present in a folder (called `foldername` below) and optionally its subfolders. A Python script called `default.py` must be in the root of this folder and will be the main script that starts the application.

```
ensymble.py py2sis --appname="your application name"
                  --caps="Local Services+NetworkServices+UserEnvironment+
                  ReadUserData+WriteUserData+SwEvent"
                  --uid="0xE0000000"
                  --version=2.0.1
                  -v ./foldername
```

For application installation packages created in this way to work, it is necessary to pre-install the Python interpreter separately. However, Ensymble also has the facility to take the output of this command and create a new SIS file which also embeds a Python interpreter. This could be the official release or one you've built yourself using the Python for S60 Community Edition described in Section 2.5. The Ensymble command for this is `mergesis`; see the Ensymble documentation for details.

#### 3.5.2 How to sign your SIS for testing

The best way to test your application on device is to package it as a Symbian installation file. Once packaged, you have to sign the file due to the security restrictions of the Symbian platform.

Open Signed Online will work for developing applications on your test device and Developer Certificates are not necessary. The Open Signed Online service provided by Symbian gives you

the ability to sign SIS files during application development and prototyping; it is not intended to allow commercial distribution. To use the Open Signed service you have to own the SIS you are signing.

Full instructions are available on the Symbian Signed web site; go to the Symbian Signed, Open Signed Online service ([www.symbiansigned.com/app/page/public/openSignedOnline.do](http://www.symbiansigned.com/app/page/public/openSignedOnline.do)) and get your SIS file signed and ready to test on-device.

## 4 Conclusions

Unlike most other runtimes, you can never say, 'Python can't do it,' since Python (with a bit of work) can do all the same things as native Symbian C++ applications. Unfortunately there isn't always an off-the-shelf solution. We hope this paper has given helpful pointers for how to proceed in such a situation.

More information about extending Python is available in the official Python documentation, at [python.org/doc](http://python.org/doc). Symbian-specific examples can be found in the PyS60 source code and PyS60 developer manual. An example that uses Cog for interface generation is available in the open source LogMan project ([code.google.com/p/logman-for-symbian/](http://code.google.com/p/logman-for-symbian/)). Python for S60 Community Edition eases hybrid language development and brings many extensions under one project umbrella.

The Flyer Framework provides a free and simple way to add a Flash UI to a PyS60 application, or to extend Flash Lite applications with native platform features. For Python developers, Flyer gives you one way of building a richer graphical interface. On the other hand, for Flash developers it is the only extension mechanism which is portable across multiple platforms but also flexible enough to allow access to all native features on any platform.

Nokia engineers have revealed that they are porting Python version 2.5.1 to S60 via an unofficial announcement on the Forum Nokia discussion boards<sup>15</sup> (where there is a very active developer community). This port has been enabled by the P.I.P.S. and Open C projects, which will also enable the porting of many more popular Python modules from the desktop. With Flyer, all of this functionality can also be made available to Flash Lite developers. Python appears to offer the promise of quick and easy solutions to a lot of mobile programming issues in the future, including providing the glue between other environments in a multi-language programming project. However, without further work on interface generators for Symbian C++, quite a lot of manual work is still required to create extensions. That said, if the Python interpreter was included in the firmware of Symbian devices, its popularity would be likely to soar.

In the fourth and final paper in this series, Bernd Wiegmann throws another runtime into the mix by discussing Widgets and ways of extending them on Symbian OS.

---

<sup>15</sup> See [discussion.forum.nokia.com/forum/showthread.php?t=124442](http://discussion.forum.nokia.com/forum/showthread.php?t=124442) and later discussions