F O R U M   N O K I A

Programming with Python for Series 60 Platform

Version 1.2; September 28, 2005

Python for
Series 60 Platform

NOKIA

Contents

Change History

December 10, 2004	Version 1.0	Initial document release.
June 29, 2005	Version 1.1.5	Sections 1.3, 10, 12, 16, Appendix J, Appendix K, and Appendix L added. Sections 1, 4.2.1, 11, and 14 updated.
September 28, 2005	Version 1.2	Sections 1.3, 12, 13, 14 16.1, Appendix I, and Appendix L updated.

1 Introduction

The Python for Series 60 Platform (Python for Series 60) execution environment simplifies application development and provides a scripting solution for the Symbian C++ APIs. This document is for Python for Series 60 release 1.2 that is based on Python 2.2.2.

The documentation for Python for Series 60 includes three documents:

- *Getting Started with Python for Series 60 Platform [1]* contains information on how to install Python for Series 60 and how to write your first program
- *Python for Series 60 Platform API Reference [2]* contains API and other reference material
- This document contains code examples and programming patterns for Series 60 devices that can be used as basis for programs. It is recommended that the sections in this document be read in the order presented

Python for Series 60 as installed on a Series 60 device consists of:

- **Python** execution environment, which is visible in the application menu of the device and has been written in Python on top of Python for Series 60 Platform (see *Series 60 SDK Help* documentation *[3]*)
- Python interpreter DLL
- Standard and proprietary Python library modules
- Series 60 UI application framework adaptation component (a DLL) that connects the scripting domain components to Series 60 UI
- Python Installer program for installing Python files on the device, which consists of:
    - Recognizer plug-in
    - Symbian application written in Python

Tip: The Python for Series 60 developer discussion board [5] on the Forum Nokia Web site is a useful resource for finding out information on specific topics concerning Python for Series 60. You are welcome to give feedback or ask questions about Python for Series 60 through this discussion board.

1.1 Scope

This document includes practical examples on programming with Python for Series 60. The sample programs can be used as a basis for the users' own programs.

1.2 Audience

This guide is intended for developers looking to create programs that use the native features and resources of the Series 60 phones. The reader should be familiar with the Python programming language (see http://www.python.org/) and the basics of using Python for Series 60 (see *Getting Started with Python for Series 60 Platform [1]*).

1.3 New in Release 1.2

This section lists the updates in this document since release 1.1.6.

- There are some general updates in Section 16.1, *Required Modifications to the Example Extension*.
- Chapters 12, *Real-Time Graphics Support and Key Event Handling: ball.py*, Appendix I, *Source Code for default.py*, and Appendix L, *Source Code for Ball* contain some code updates.
- Chapter 14, *Making Stand-Alone Applications from Python Scripts* contains updated `py2sis` and SVG icon information.

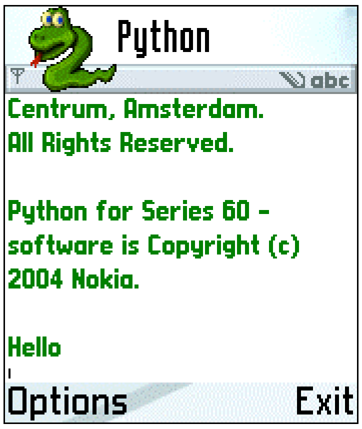
1.4 Typographical Conventions

The following typographical conventions are used in this document:

Bold	Bold is used to indicate windows, views, pages and their elements, menu items, and button names.
Italic	Italics are used when referring to another chapter or section in the document and when referring to a manual. Italics are also used for key terms and emphasis.
`Courier`	Courier is used to indicate parameters, file names, processes, commands, directories, and source code text.
>	Arrows are used to separate menu items within a path.

2 The Hello World Application

The shortest possible application written in Python prints "Hello" on the console. The application `hello.py` consists of the line `print "Hello"` to produce the result displayed in Figure 1.



Centrum, Amsterdam.
All Rights Reserved.

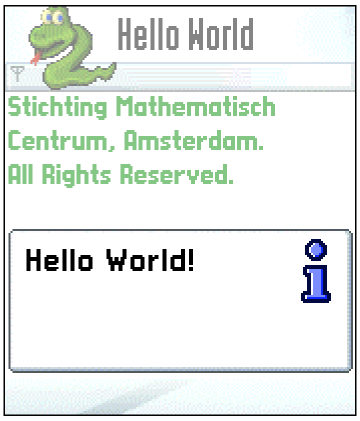
Python for Series 60 -
software is Copyright (c)
2004 Nokia.

Hello

Figure 1: Hello

A graphical version of the Hello World application code has three lines and creates the output displayed in Figure 2.

```
import appuifw
appuifw.app.title = u"Hello World"
appuifw.note(u"Hello World!", 'info')
```



Stichting Mathematisch
Centrum, Amsterdam.
All Rights Reserved.

Hello World!

Figure 2: Hello World!

The first line makes the application UI framework (`appuifw`) available to the script. The constant `app` is predefined and it represents the application. It has a Unicode string attribute `title` that can be set to change the title of the application. The attribute can be read and stored like any other variable:

```
old_title = appuifw.app.title
# Import and run some other application here that changes the title
...
# Restore the title
appuifw.app.title = old_title
```

The final statement `appuifw.note(u"Hello World!", 'info')` creates and displays a note (see Figure 2). The text to be displayed must be Unicode because the Symbian platform uses Unicode, but the note type is a plain string. The two other note types are `error` and `conf`.

3 Using the Bluetooth Console

Bluetooth console is a Python application that can be used as any Python application. Bluetooth console is the easiest way to run Python on a phone, assuming that you have a Bluetooth connectivity in your PC.

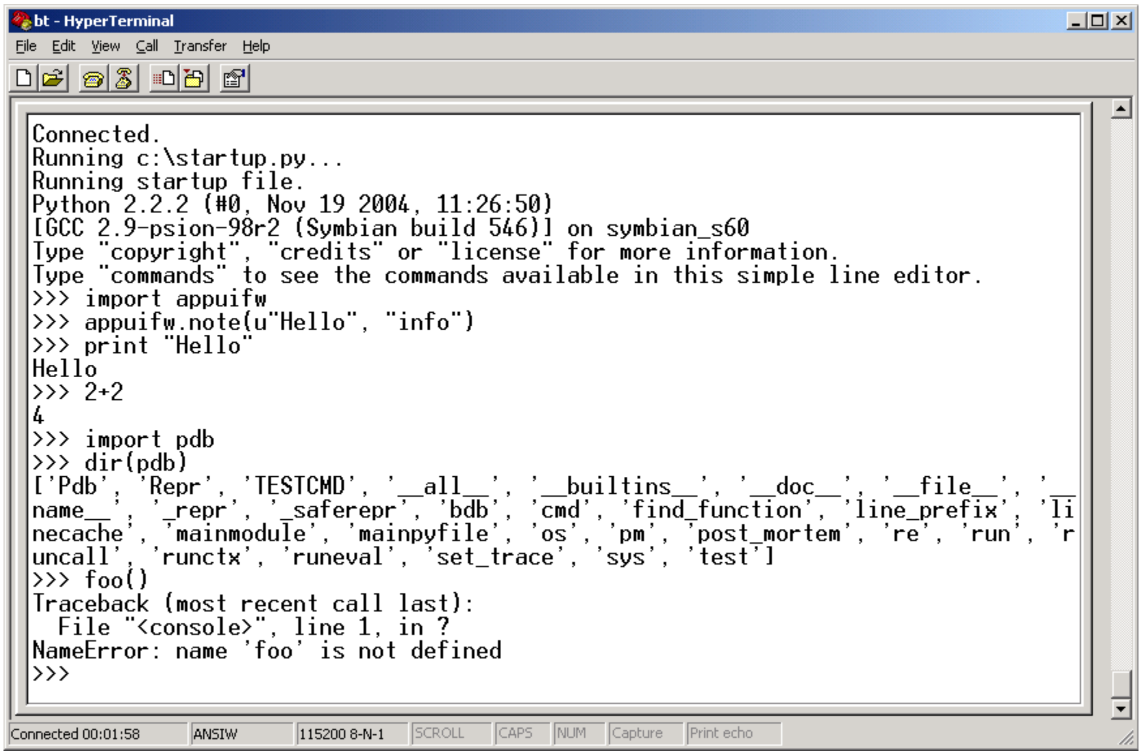
A listening Bluetooth RFCOMM serial port is required on your PC. Consult the documentation of your Bluetooth device for instructions. You can use any VT100-compatible serial terminal software to communicate with the phone. The standard Windows HyperTerminal is also adequate. Connect the terminal emulator to the port assigned to the Bluetooth serial service.

Note that the Bluetooth console application accepts both CR and LF as a line termination character, and that the CR-LF combination is interpreted as two line terminations. To verify that HyperTerminal sends only CR and not CR-LF, check that the option **File > Properties > Settings > ASCII Setup > Send line ends with line feeds** is not selected.

To start the Bluetooth console, choose **Options > Run Script** and start `bt_console.py`. When the Bluetooth console starts for the first time, the phone searches for nearby Bluetooth devices and prompts you to choose a device and a port on that device. An option allows you to save the device address for later connection without performing the time-consuming device discovery process.

Note that the device discovery fails when there is a Bluetooth connection open. For example, this can happen if you have just sent a file over a Bluetooth connection. To disconnect all Bluetooth connections, disable and re-enable the Bluetooth connection from the phone. The discovery of devices is the only operation that cannot be done when there is an open connection – connecting to a previously discovered default host works even if there are other open connections.

Figure 3 contains sample Bluetooth console content from your PC. Everything that you type on the console is sent to the phone and interpreted, and everything printed is sent back. For details of this process, see Chapter 8, *Bluetooth Sockets*.



```
Connected.
Running c:\startup.py...
Running startup file.
Python 2.2.2 (#0, Nov 19 2004, 11:26:50)
[GCC 2.9-psion-98r2 (Symbian build 546)] on symbian_s60
Type "copyright", "credits" or "license" for more information.
Type "commands" to see the commands available in this simple line editor.
>>> import appuifw
>>> appuifw.note(u"Hello", "info")
>>> print "Hello"
Hello
>>> 2+2
4
>>> import pdb
>>> dir(pdb)
['Pdb', 'Repr', 'TESTCMD', '__all__', '__builtins__', '__doc__', '__file__', '__
name__', '_repr', '_saferepr', 'bdb', 'cmd', 'find_function', 'line_prefix', 'li
necache', 'mainmodule', 'mainpyfile', 'os', 'pm', 'post_mortem', 're', 'run', 'r
uncall', 'runctx', 'runeval', 'set_trace', 'sys', 'test']
>>> foo()
Traceback (most recent call last):
  File "<console>", line 1, in ?
NameError: name 'foo' is not defined
>>>
```

```
Connected 00:01:58   ANSIW   115200 8-N-1   SCROLL   CAPS   NUM   Capture   Print echo
```

Figure 3: Bluetooth console

The scripts installed in the `\system\apps\python\my` directory are added to the command history by default, so you can run them in the console with the **Ctrl+p/Ctrl+n** keys.

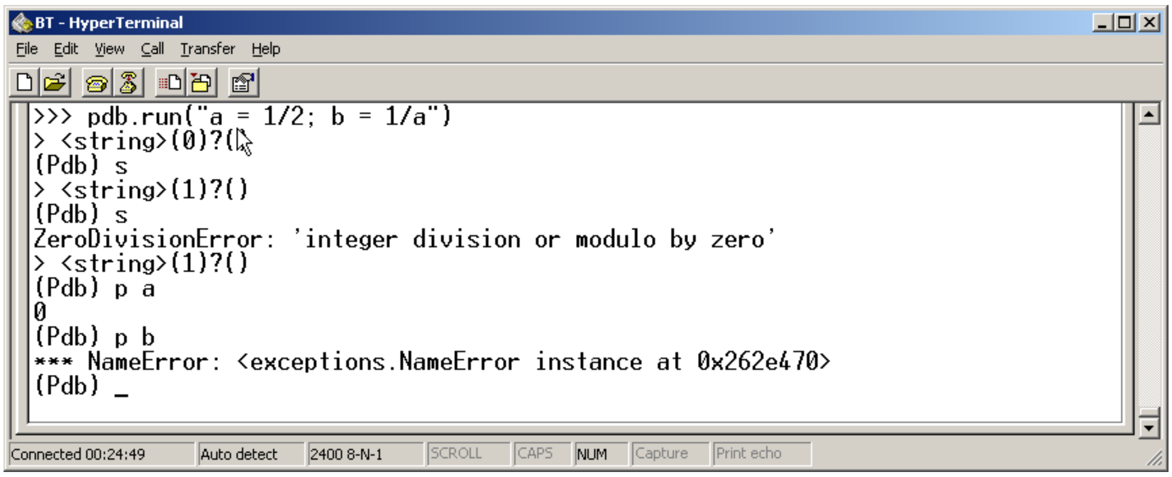
The default line editor is quite limited. If you have installed a more advanced line editor, such as the `Pyrepl` library, you can start that from the command line.

Typing

```
import appuifw
appuifw.note(u"Hello", 'info')
```

has the same effect as the example in Figure 3: a note saying "Hello" is displayed on the phone. `print "Hello"` prints "Hello" on the console. 2+2 evaluates a mathematical expression. The last part of the example in the figure, starting with `Traceback`, shows a useful feature of the Bluetooth console: all exception information is shown in the console. This is important when a running GUI application covers the default text console of the phone.

One essential application that can be run over the console is the Python debugger `pdb`. This application is not automatically installed (see *Getting Started with Python for Series 60 Platform [1]* for more information). Figure 4 shows how `pdb` allows you to use single-step expressions (command **s**) and query the values of variables (command **p**).



```
>>> pdb.run("a = 1/2; b = 1/a")
> <string>(0)?()
(Pdb) s
> <string>(1)?()
(Pdb) s
ZeroDivisionError: 'integer division or modulo by zero'
> <string>(1)?()
(Pdb) p a
0
(Pdb) p b
*** NameError: <exceptions.NameError instance at 0x262e470>
(Pdb) _
```

Figure 4: Python debugger

3.1 TCP/IP Console

If Bluetooth wireless technology is unavailable, you can also use the console over any TCP/IP transport service your phone supports, such as GPRS, EDGE, or UMTS. To run the console over TCP/IP, first set up a listening TCP port on a host the phone can access. For example, if you have the "Netcat" utility on a Linux machine, you can use the following commands to receive the connection (replace 1025 with the port you want to use):

```
  stty raw -echo; nc -l -p 1025; stty sane
```

Then you can run the console over TCP/IP with the following script in the device:

```
import btconsole
from socket import *
sock=socket(AF_INET,SOCK_STREAM)
sock.connect(("address of listening host",1025))
btconsole.run_with_redirected_io(sock,btconsole.interact,
                    None, None, locals())
sock.close()
```

4 GUI Programming

This chapter introduces GUI programming with Python for Series 60. It starts with a simple example application and finally presents a pattern for writing more complex GUI applications.

4.1 First Example: Weather Maps

Figure 5 displays a simple application with a graphical user interface.

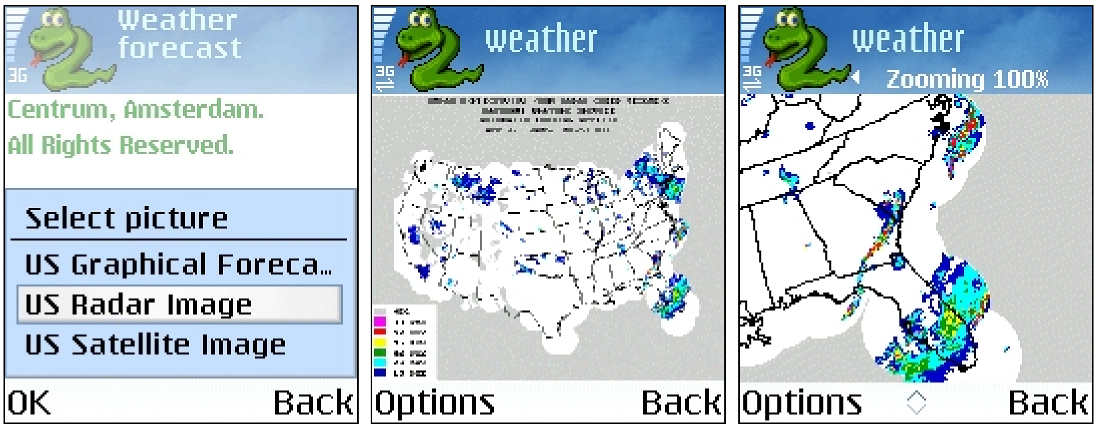


Figure 5: Weather forecast

The application allows users to select what information to fetch, and displays that information with a suitable application.

For the source code, see *Appendix A*, *Source Code for Weather Maps*.

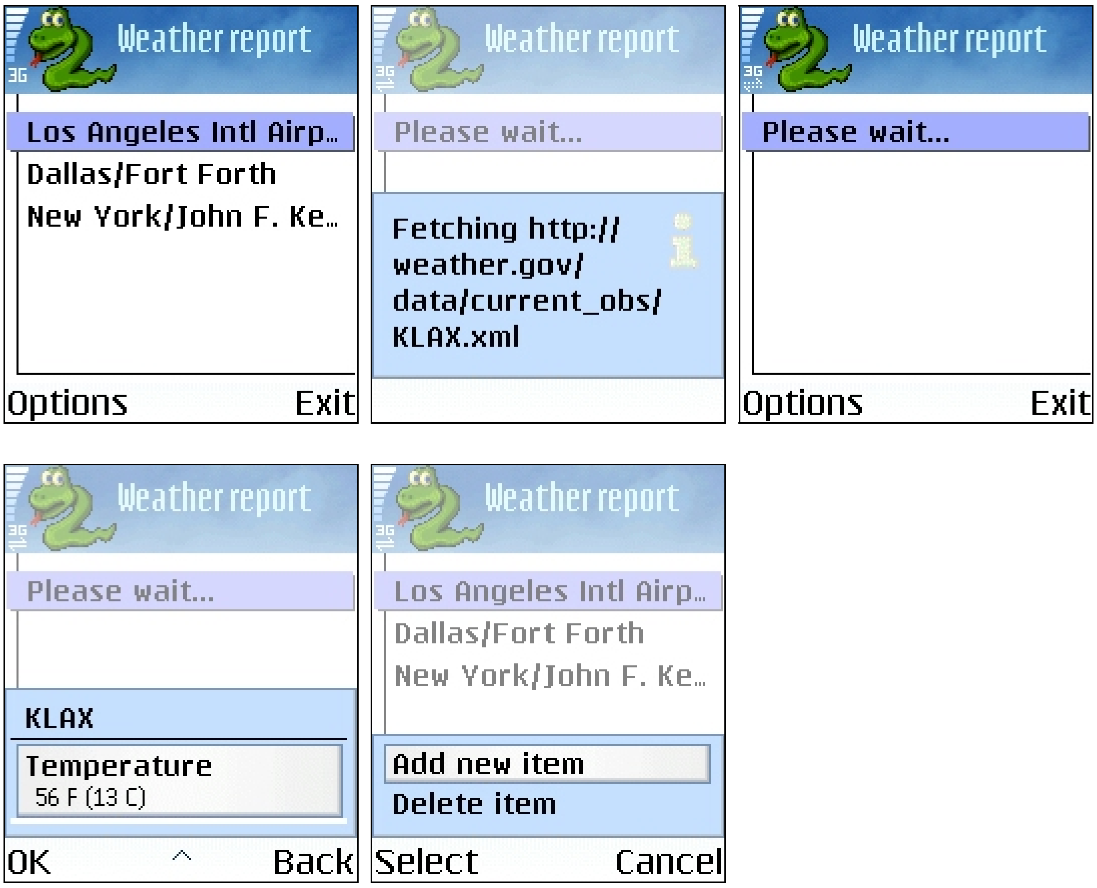
The example is single-threaded and uses a dialog, which simplifies the application design. `Popup_menu` returns the index if the users select a valid value, and `None` in other cases.

`urllib` is a standard library module that enables users to retrieve Web content based on URLs by using the `urlretrieve` function. In this example, the image (PNG, GIF, or JPEG file) is stored in a temporary file with the corresponding extension (`.png`, `.gif`, or `.jpg`). Note that the images may be quite large and may therefore take up a large amount of memory. A content handler object displays the content appropriately on the phone. If you modify the program by adding more content types, the extensions can be any of those the phone recognizes.

The application outputs to the console and thus everything is printed on the default screen, which is on the background of the application. In the example in Section *4.2*, *Second Example: Weather Information*, this limitation has been fixed. If there is an error in the application, the failure information is directly visible. In other cases, it is recommended to redirect the error information to a separate file. For more information, see Chapter *7*, *Logging*.

4.2 Second Example: Weather Information

Figure 6 shows a more comprehensive example that creates a `Listbox` with three choices. Selection of a choice starts the downloading of the appropriate file. The application indicates which file is being retrieved. The `Listbox` contents are replaced by the text **Please wait…** for simplicity. After receiving the information, the application shows it in a two-line `popup_menu`. The left softkey also works (see the last image of Figure 6), but the selection of an option has not been implemented in this example.


Weather report
Los Angeles Intl Airp...
Dallas/Fort Forth
New York/John F. Ke...
Options Exit

Weather report
Please wait...
Fetching http://weather.gov/data/current_obs/KLAX.xml

Weather report
Please wait...
Options Exit

Weather report
Please wait...
KLAX
Temperature
56 F (13 C)
OK Back

Weather report
Los Angeles Intl Airp...
Dallas/Fort Forth
New York/John F. Ke...
Add new item
Delete item
Select Cancel

Figure 6: Weather report

For the source code, see *Appendix B, Source Code for Weather Information*.

After the setup information, the `find_value` function is defined. It looks for the value between `<tag>` and `</tag>` in text. This function always returns something: if a value does not exist, the function returns an empty string.

A more robust and extensible way to implement this would be to use a parser that produces a SAX or DOM tree. However, in this case a hand-written XML parser is adequate.

Four handlers are then defined: `handle_selection` is for handling the event of selecting an item in `Listbox`. This is passed to the `Listbox` when it is created. The `handle_add` and `handle_delete` functions are placeholders for the events of adding new locations and deleting existing ones. The last handler is for the exit key.

The `Listbox` is first created and then set to be the application body. This makes it visible to the users. Initially, the `Listbox` shows the airports of the three cities, but at the beginning of the `handle_selection` function it is replaced by the text **Please wait...** The original text is restored at the end of `handle_selection`.

Since `Listbox` is a UI control and not a dialog, creating a listbox does not cause the application to wait for input. When the application is run from the Python execution environment (see Chapter *13, Python Execution Environment: default.py and Others*), this would cause the execution to fall back to the execution environment implemented by the script `default.py`. For a solution to this problem and a discussion on the difference between scripts installed as stand-alone applications and scripts run from the Python execution environment, see Section *4.2.1*, *Active objects*.

The following lines use `urlretrieve` to get a document and save it to a temporary file:

```
urllib.urlretrieve(weather_url, tempfile)
f = open(tempfile, 'r')
weatherinfo = file.read()
f.close()
```

This file is then opened, and read into `weatherinfo`.

The line `weather = find_value(weatherinfo, "weather")` reads `weatherinfo`, finds `<weather>` and `</weather>`, and assigns the text between these to `weather`.

Note that all UI elements use Unicode as the default encoding. Therefore,

```
[(u"Weather", unicode(weather)), (u"Temperature",
unicode(temperature_string))]
```

is written as a list that is passed to the `popup_menu`. Since this is a list of pairs, the UI element is the two-line pop-up menu.

One weakness of this sample application is that although the text in the `Listbox` changes, the handler is not changed. Users might select the text **Please wait...**, and as a result, the program would display the weather information for Los Angeles, which is not desired. A fix for this would be to use a variable to keep track of the state of the `Listbox`, so that it could be known what the text was when the users selected something. A better solution would be to use the `bind` method to reset the key binding in the handler function as follows:

```
lb.bind(key_codes.EListboxEventEnterKeyPressed, None)
```

4.2.1 Active objects

Symbian threads and active objects work roughly as presented in Figure 7: each thread can contain a number of active objects. Note that active objects are co-operative and non-preemptive. As a Python programmer, you typically need to take care of active objects as they relate to UI programming, and sockets as described in Section *4.3*, *Application Skeleton*.

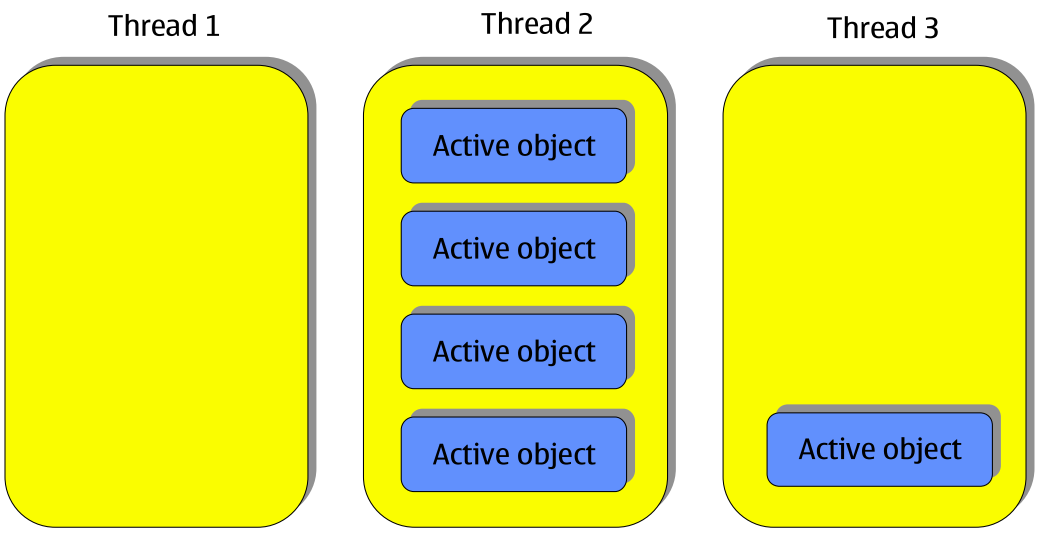


Figure 7: Symbian threads and active objects

All Symbian applications have a single thread that takes care of user interaction. This is also the main thread in Python applications, which allows you to directly call UI elements from Python. The downside is that if the main thread is blocked, the UI is not updated. Although

```
import thread
l = thread.allocate_lock()
```

could be used, this would block the whole application. The `e32.Ao_lock` facility enables solving this problem, as it allows active objects — in this case, the UI event handlers — to run while the application waits in the lock.

An instance of `e32.Ao_lock` can be used to keep an application that is run from Python execution environment script from falling back too early. This is achieved by creating a lock instance, setting `app.exit_key_handler` to signal the lock, and then blocking in a wait for the lock. A setup like this is not necessary for scripts installed as stand-alone applications as the application only exits when the **Exit** key is pressed. For information on scripts installed as stand-alone applications, see Chapter *13*, *Python Execution Environment: default.py and Others*.

Warning: In non-trivial applications, it is almost always necessary to use an `Ao_lock`. Otherwise, an UI application will exit before the user has a chance to see anything on the screen. Also, special care must be taken to arrange the application logic in such a way that it is not possible to escape from the application without signaling the eventual `Ao_lock` waiter. Otherwise, the active object-handling framework gets into an erroneous state and may not function properly.

4.3 Application Skeleton

Figure 8 shows two collaborating threads, the other with active objects.

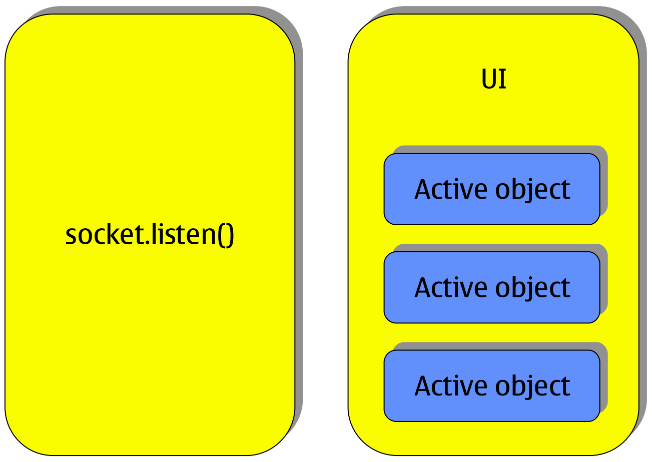


Figure 8: Two collaborating threads

In many Internet applications, users interact with an application that is also listening to a socket. When the application receives new information from the Internet, it needs to update the interface immediately. The thread listening to the socket cannot be allowed to update the UI, because this would require careful coordination with the UI thread, and might crash the program. The `e32.is_ui_thread` tells whether the current thread is handling the UI and whether it is safe to do something with the program.

The sample application, which has been edited for clarity but is based on a real application, is the UI for a to-do list on a network. When started, the application asks users to which server it should connect. The application receives the task list from the network and allows the users to add and delete tasks.

For the source code, see *Appendix C*, *Source Code for Application Skeleton*.

Starting from the bottom of the example, the Python idiom

```
if __name__ == "__main__":
    main()
```

has been used. This makes it possible to use the same program both as an application and as a library: importing a program as a library in the interactive console facilitates debugging it.

There is also a function called `main`. It creates an instance of class `MyApp` and then enters its `loop` method, which acts as the main event loop for the application. Finally, the `close` method of the application is called to allow for cleanup. In this case, the `close` method clears the application menu, body, and the exit key handler.

After the imports, the `e32.ao_yield` function makes sure that the UI is refreshed after the potentially lengthy process of importing libraries. The operating system assumes that an application that does not handle its UI events for a long time is broken, and tries to close it.

The constructor for the `MyApp` class creates a lock and an `exit_flag` variable that is initially set to `False`. After this the constructor does some application-specific preparations.

The `loop` method implements the main loop. It first reserves the lock (which was free when created) and then goes into a loop that is completed by changing the `exit_flag` variable to `True`. The `abort` function is the only place where the exit flag is changed to `True`. When the exit flag changes, the loop function returns and the `close` method is called in the `main` function.

The `loop` method calls the `refresh` method, waits on the lock, and repeats until told to stop. The `refresh` method is the place to put the code that is needed for updating the UI after an event has been processed.

When the database has been affected by any thread in this or some other application, the `notify` method is called. Calls to this method are not shown in the example code. The `notify` method can also be safely called from outside the UI thread because it only signals the internal lock. Other threads can also change common data and call the `notify` method. When the `MyApp` execution reaches a suitable point, the `refresh` function gets called, notices the new data, and updates the UI.

Methods `handle_modify`, `handle_add`, and `handle_delete` are for modifying, adding, and deleting data in the database. Database information is stored in `self.data` so that it does not need to be fetched from the database every time.

One optimization not implemented here would be to use a flag to keep track of whether anything has changed in the user-visible view. If not, the `refresh` function does not need to be called and the `loop` function can go directly back to wait.

For a practical example on this kind of application structure, see Chapter *11*, *Handling Key Bindings: RSS Reader*.

5 Send SMS and Tabbed View

This chapter presents a sample application for sending an SMS message and creating a tabbed view application (see Figure 9).

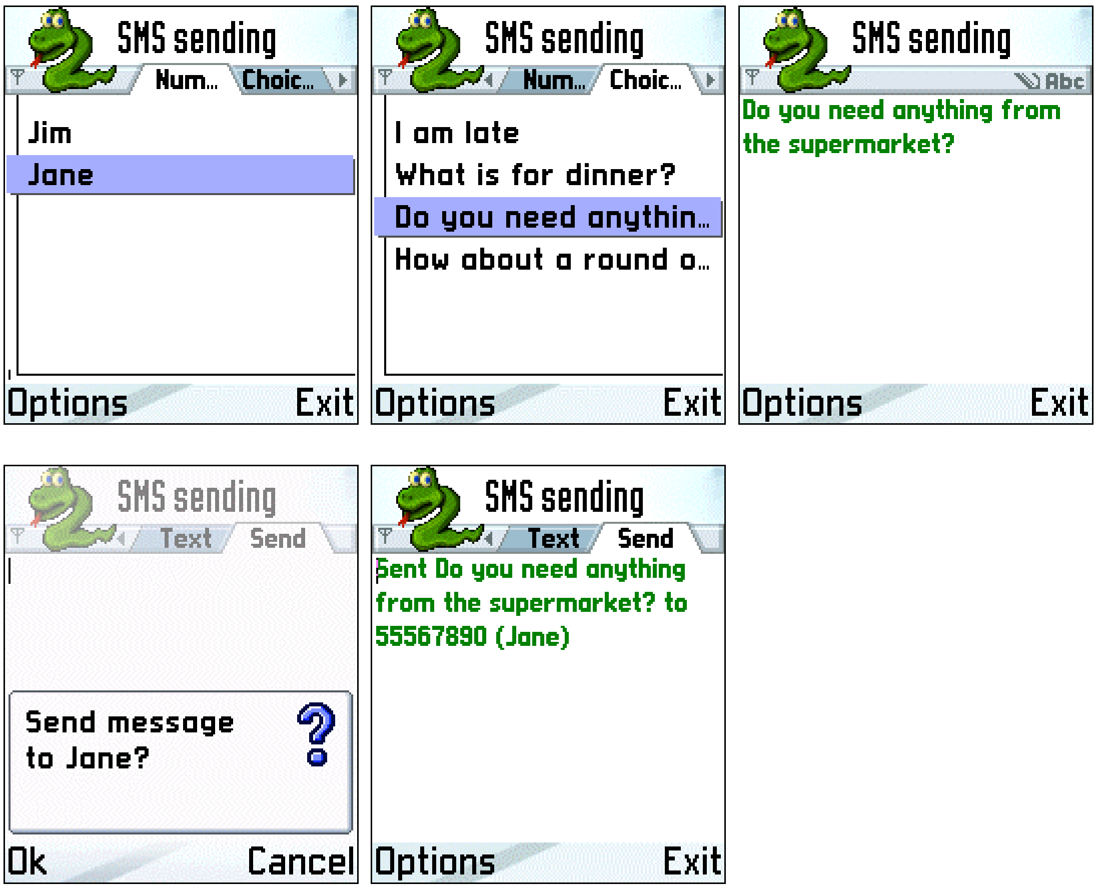
For the source code, see *Appendix D, Source Code for SMS Sending*.

The sample application is a utility for sending SMS messages.

To send a message:

1. Select a recipient from a pre-defined list.
2. Select a stock message.
3. View the stock message.
4. Send the message.

Each action takes place in a separate tab view. Users are not allowed to edit the stock message. The last tab is a log of all messages sent during the session.



Jim
Jane

I am late
What is for dinner?
Do you need anythin...
How about a round o...

Do you need anything from the supermarket?

Send message to Jane?

Sent Do you need anything from the supermarket? to 55567890 (Jane)

Figure 9: SMS sending

The statement in the `activate` method of the `SendView` class

```
messaging.sms_send(nbr, txt)
```

sends the SMS.

The variable `nbr` is the phone number (an ASCII string) to which the SMS is sent, and the `txt` variable is the text to be sent (Unicode). There is no automatic confirmation query, so you would need to write one if desired.

The main application instantiates the `SMS_multiviewApp` object and calls its `run` method. This method only initiates the tabbed view and then waits in an `Ao_lock`. The `close` method performs a cleanup. The constructor for `SMS_multiviewApp` creates the different views in the tabbed view as instances of different classes, and sets the tabs.

The `handle_tabs` method is called when the users change tabs with the navigation key. Note that there are other ways to change the view (using `activate_tab`), but in those cases `handle_tabs` needs to be explicitly called. The `handle_tabs` method calls the `activate` method of the appropriate view instance.

The structure of the applications allows all view objects (`NumbersView`, `ChoiceView`, `TextView`, and `SendView`) to know about the `SMS_multiviewApp` object and the `SMS_multiviewApp` to know about all the view objects. When a view object needs to know something that is maintained by another view, it asks the `SMS_multiviewApp` object. Therefore, the `SMS_multiviewApp` object has the necessary methods for passing information.

Because all views have the same structure, a `ChoiceView` instance can serve as an example. It presents the stock messages as a list. Since some messages may be too long to show on one line, the next tab shows the full contents. When created, a `ChoiceView` instance registers the `SMS_multiviewApp` object it was initialized with, creates the texts for display, and instantiates a listbox.

When activated, a `ChoiceView` instance sets the body to be the `Listbox` and resets the menu choices. `Listbox` and the `Text` UI control look different in tabbed view applications, as can be seen in Figure 9: a `Listbox` leaves the tabs visible but a `Text` takes over the full screen. The events generated by left and right arrow keys are always consumed by the main application, which moves between views. This means that users cannot edit stock messages in this application because left and right arrow keys are not available for this purpose.

The `handle_select` method has no real practical use and is included mainly as an illustration and to provide comfort to the users. The `get_text` method fetches the text corresponding to the selection. One option is always current in the `Listbox`; this can be specified in the constructor and is the first one by default. If a user selects "Jane" from the list, moves to another view, and returns, "Jane" is still selected.

The `handle_send` method takes care of the **Send** menu selection. The method moves directly to the **Send** tab, which is a two-stage process: `activate_tab(3)` changes the view and selects the active tab, but does not call any functions. The `handle_tab(3)` method is called from `SMS_multiview`. `SendView` gets the selected text from `ChoiceView` by calling `get_text` from `SMS_multiview`, which calls `get_text` from `ChoiceView`.

If you want to try the application yourself, add some phone numbers in `NumbersView`, uncomment **Import messaging**, and uncomment `messaging.sms_send` in `SendView`. It is better to debug the sample application when it does not send real SMS messages.

6 Access to File System

File system access with Python for Series 60 works the same way as in the Linux or Windows operating systems. For example, the following line opens a file for reading:

```
f = open("c:/temp/test.txt",'r')
```

In the mode field, 'r' is for reading, 'w' is for writing, 'a' is for appending, and 'r+' is for both reading and writing.

Note that you can use forward slashes ("/") as file path separators. If you use backslashes ("\"), you must write the line above as

```
f = open("c:\\temp\\test.txt",'r')
```

The default "current working directory" in Python for Series 60 is the Z-drive, which is in ROM. Therefore,

```
f = open("test.txt",'w')
```

does not work since the system tries to store `"test.txt"` in read-only memory. Typically, the drive letters refer to the following devices:

- 'C' is the built-in phone memory
- 'D' is a RAM disk
- 'E' is an extra memory card
- 'Z' is read-only ROM memory

6.1 Example: File Browser

Figure 10 shows the output of a simple File Browser application.

For the source code, see *Appendix E, Source Code for File Browser*.

The technique described in Section *4.2*, *Second Example: Weather Information* has been used to arrange a clean exit out of the script application in the script execution environment.

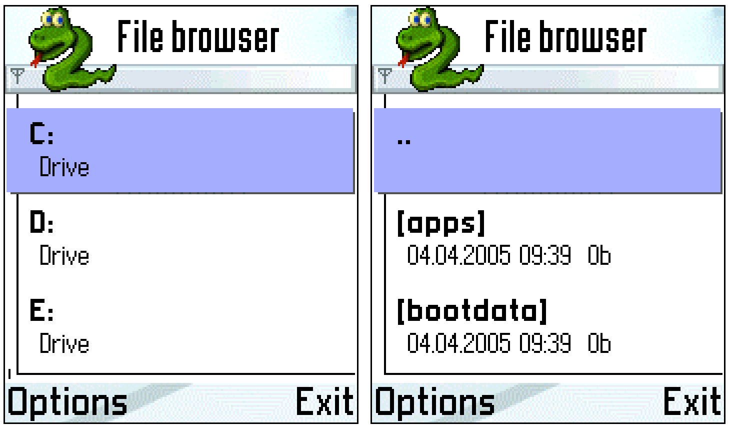


Figure 10: File browser

7 Logging

Python refers to file objects corresponding to the standard input, output, and error streams as `stdin`, `stdout`, and `stderr`. These variables are in the `sys` module. Standard output captures all text that a program prints, whereas standard error captures all error messages. The `stdout` and `stderr` variables can be assigned to any objects that have a `write` method. For example, if you want your program to write all error messages to a file called `errormessages.txt` in C drive, you can do the following:

```
import sys
errorfile = open("c:/errormessages.txt", "w")
sys.stderr = errorfile
…
errorfile.close()
```

If you want, you can do a redirection temporarily and restore later:

```
import sys
errorfile = open("c:/errormessages.txt", "w")
old_stderr = sys.stderr
sys.stderr = errorfile
# Some code here …
sys.stderr = old_stderr
errorfile.close()
```

If you want to run a test script while output is redirected to the Bluetooth console, use the `btconsole` module functions. The `btconsole.connect` function connects to a Bluetooth serial port and returns a socket that represents that connection. This is the same function the Bluetooth console application uses for making connections, which means it supports preserving the default host. The `btconsole.run_with_redirected_io` function inside the `btconsole` module redirects the `stdin`, `stdout`, and `stderr` variables of a given function to a given socket. If run inside the UI thread, the `btconsole.run_with_redirected_io` function also installs its own exit key handler, which closes the socket if the **Exit** key is pressed. This can be used to abort execution of the script in question.

For example (run this script with the **Run script** menu option, not from the Bluetooth console):

```
def print_stuff(message):
  print "Attention: %s"%message

print_stuff("This goes to the normal text console.")

import btconsole
sock=btconsole.connect() # Connect to a Bluetooth serial port
btconsole.run_with_redirected_io(sock,print_stuff,
         "This goes to the Bluetooth serial port.")
sock.close()
```

7.1 A Logger Module

It is customary for operating systems to cache writings to the file system. The `flush` method is used to make sure that the file buffer is written to the disk. You can do the following in order to get debugging information to a file even if the program you are debugging crashes the Python execution environment:

```
import sys
debugfile = open("c:/system/debuginfo.txt","w")
sys.stdout = debugfile
# Code...
# This will now go to the file
print "Execution reached this far"
```

```
sys.stdout.flush()
```

On the last line, `sys.stdout.flush` calls `debugfile.flush` after the redirection. Conveniently, `sys.stdout.flush` has also been defined for standard `stdout`, although it is practically empty. This means that the lines that handle `debugfile` can be commented out.

However, even flush can fail to write bits to the hard drive. This may be the case if you are debugging on the emulator and it crashes. The following code is the best way to make sure that everything has been written to a file (it also works for `stderr`):

```
class Logger:
    def __init__(self, log_name):
        self.logfile = log_name

    def write(self, obj):
        log_file = open(self.logfile, 'a')
        log_file.write(obj)
        log_file.close()

    def writelines(self, obj):
        self.write(''.join(list))

    def flush(self):
        pass
```

To use this, create a file `logger.py`, insert the code above into it, install the file as a library module (see *Getting Started with Python for Series 60 Platform [1]*), and do the following:

```
import sys
import logger
my_log = logger.Logger("c:/system/my_log.txt")
sys.stderr = sys.stdout = my_log
print "Testing logging"
```

On the last line, print **Testing logging** causes the `write` method of the `Logger` class to be called. This opens the file `c:/system/my_log.txt` for appending, writes the text there, and then closes the file.

A `logger` module could also contain the following convenience function – whose implementation has been taken from the `code` module – for writing as much information about an exception trace as possible.

Note: The `logger` module is not part of the Python for Series 60. You can use the module by copying the code in this document.

```
def print_exception_trace(filename):
    import sys, traceback
    logfile=open(filename,'a')
    try:
        type, value, tb = sys.exc_info()
        sys.last_type = type
        sys.last_value = value
        sys.last_traceback = tb
        tblist = traceback.extract_tb(tb)
        del tblist[:1]
        list = traceback.format_list(tblist)
        if list:
            list.insert(0, "Traceback (most recent call last):\n")
        list[len(list):] = traceback.format_exception_only(type, value)
    finally:
        tblist = tb = None
    map(logfile.write, list)
    logfile.close()
```

The standard traceback module of the Python is applied to retrieve and format the information. The map function on the last line calls logfile.write with all members of list.

To use this function, enclose your code in a try block and call print_exception_trace if something goes wrong:

```
from logger import print_exception_trace
    try:
        # Suspicious code here
        …
    except:
        print_exception_trace("c:/errorlog.txt")
```

Even better, print_exception_trace could be made a method of the Logger class.

8 Bluetooth Sockets

Bluetooth sockets behave quite similarly to normal Internet sockets. The sample Bluetooth console described in this chapter behaves quite like the built-in Bluetooth console. The idea of the console is that everything that a user types at a remote console is transmitted to a Python instance running on the phone, and all replies are sent back.

For the program code, see *Appendix F, Source Code for Bluetooth Sockets*.

```
class socket_stdio:
    def __init__(self,sock):
        self.socket=sock
    def read(self,n=1):
        return self.socket.recv(n)
    def write(self,str):
        return self.socket.send(str.replace('\n','\r\n'))
    def readline(self,n=None):
        buffer=[]
        while 1:
            ch=self.read(1)
            if ch == '\n' or ch == '\r':   # return
                buffer.append('\n')
                self.write('\n')
                break
            if ch == '\177' or ch == '\010': # backspace
                self.write('\010 \010') # erase character
                                        # from the screen...
                del buffer[-1:] # ...and from the buffer
            else:
                self.write(ch)
                buffer.append(ch)
            if n and len(buffer)>=n:
                break
        return ''.join(buffer)
    def raw_input(self,prompt=""):
        self.write(prompt)
        return self.readline()
    def flush(self):
        pass
```

The functionality is provided by the `socket_stdio` class, which defines methods similar to the standard input and output methods. Everything is read from a socket and written to a socket.

```
sock=socket.socket(socket.AF_BT,socket.SOCK_STREAM)
# For quicker startup, enter here the address and port to connect to.
target='' #('00:20:e0:76:c3:52',1)
if not target:
    address,services=socket.bt_discover()
    print "Discovered: %s, %s"%(address,services)
    if len(services)>1:
        import appuifw
        choices=services.keys()
        choices.sort()
        choice=appuifw.popup_menu(
                       [unicode(services[x])+": "+x for x in choices],
                       u'Choose port:')
        target=(address,services[choices[choice]])
    else:
        target=(address,services.values()[0])
print "Connecting to "+str(target)
```

After the `socket_stdio` class is constructed, a socket is created. The default target illustrates the format for Bluetooth addresses. If there is no default, the nearby Bluetooth devices have to be discovered. The result of the discovery is a Bluetooth address — one which the users have selected from a list of all visible Bluetooth devices — and a set of services which that Bluetooth device supports. The program displays the service keys in a pop-up menu for the users to select the final service from.

```
sock.connect(target)
socketio=socket_stdio(sock)
realio=(sys.stdout,sys.stdin,sys.stderr)
(sys.stdout.sys.stdin,sys.stderr)=(socketio,socketio,socketio)
```

The `stdin` and `stdout` variables can be connected and reconnected to the socket using the `socket_stdio` adapter class.

```
import code
try:
    code.interact()
finally:
     (sys.stdout,sys.stdin,sys.stderr)=realio
    sock.close().
```

The `code.interact` function implements an interactive console using the current `stdin`, `stdout`, and `stderr` variables. Python also uses the `interact` code function for the interactive console, which causes the Bluetooth console to work mostly the same way as the native Windows Python command line. An exception is the line editing functionality that is provided by the `socket_stdio` class, which is different from the one in the Windows operating system. After the interpreter has finished, the `stdin`, `stdout`, and `stderr` variables are returned to their original values and the socket is closed.

9 Database Access and Form

Database access is possible using two modules: `e32db` and `e32dbm`. The `e32dbm` module provides Python DBM functionality, which offers a persistent dictionary where both the keys and values are Unicode strings. That makes it possible to create a persistent dictionary, such as a `shelve` in standard Python library, using the `marshal` module to read and write Python values in a binary format.

For the source code, see *Appendix G, Source Code for Sports Diary*.

To see another example of `e32db` module, refer to the implementation of the `e32dbm` module that uses the `e32db` module internally.

One benefit of using `e32dbm` is that it allows the reading and writing of native Symbian databases. If you have several applications – some of which are written in C++ – that need to access the same database, `e32db` is the best choice. Since `e32dbm` is easier to use, it is recommended for pure Python development. The interface to an `e32db` instance is based on executing SQL statements.

Note: If you try this sample application, the database `SportsDiary.db` is created in the root folder of the C drive in the device.

The sample application is a sports diary that enables users to keep track of their training. It allows users to store the following information for an event:

- Date and time
- Length of the event in time (duration)
- Distance
- Sport (alternatives are running, skating, biking, skiing, and swimming)
- Free-text comment

It is assumed that users use the application right after they finish the sports training, so `date` and `time` default to the current date and time.

The use of several different date and time formats in different parts of the system can be confusing. A brief summary:

- Unix time is the number of seconds passed according to UTC (Coordinated Universal Time) since January 1st 1970 00:00:00 UTC. This is the default time format for the Python for Series 60 API.
- The GUI form accepts date and time values in a format derived from Unix time (Figure 11). Date is represented as the Unix time rounded down to the nearest local midnight, and time is represented as the number of seconds passed since that midnight. Both values are floats. This means that if you have a date field and a time field, you can form the Unix time they represent by simply adding the values together.
- The `e32db` SQL statements represent date and time literals in the following format: #29.04.1979 04:01:02#. The order of fields in this format depends on the current date/time format settings in use, and the SQL interpreter rejects all other formats. The `e32db.format_time` function can be used to format date/time values according to the current settings.
- The Symbian native time format is a 64-bit number, which represents microseconds since midnight of Jan 1st, 0 AD, nominal Gregorian, local time. This is the format also used internally in the `e32db` database. Normally, there is no need to worry about this format, since the `e32db` API uses Unix time by default, but in case you need it, there are also access functions for this format.

The sample application has three classes: `SportsDiaryApp`, `SportsDiary`, and `SportsDiaryEntry`. An instance of `SportsDiary` provides functions to add and delete

`SportsDiaryEntries`. The `SportsDiary` class opens a native database when it is created, and it has a `close` method for cleanup that closes the native database. If the application raises an exception, the `close` method may not be called.

The `get_all_entries` method displays the first SQL statement and it also shows how to do queries:

```
def get_all_entries(self):
    dbv = e32db.Db_view()
    dbv.prepare(self.native_db, u"SELECT * from events ORDER BY date
DESC")
    dbv.first_line()
    results = []
    for i in range(dbv.count_line()):
        dbv.get_line()
        e = SportsDiaryEntry(dbv)
        results.append(e)
        dbv.next_line()
    return results
```

In the method, `dbv` is a database view. It is first prepared with the parameters of the native database and the SQL statement `"SELECT * from events ORDER BY date DESC"`. The `first_line` method returns the first line of the resulting query. In the sample application, a list of `SportsDiaryEntries` is built by using the lines in the view.

All details about what is stored in the database are kept in the `SportsDiaryEntry` class. Therefore, the `add` and `delete` methods call the `sql_add` and `sql_delete` methods of `SportsDiaryEntry`.

`SportsDiaryEntry` starts by defining the sport. A more general application would not hard-code these options, but would rather use a table in the database to maintain a list of possible sports.

`sql_create` is the SQL statement that creates a suitable table in the database. `sql_create` is defined in the constructor of `SportsDiary`.

```
sql_create = u"CREATE TABLE events (date TIMESTAMP, duration FLOAT,
distance FLOAT, sport INTEGER, comment VARCHAR)"
```

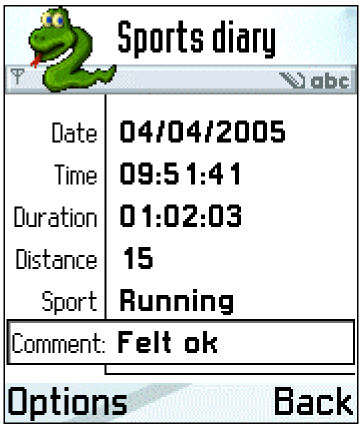
Date and time are represented as one floating-point number date. Duration, which is the length of the sports event in time, is also stored as a float, as is distance. The sport is stored as an integer, and the comment is a `VARCHAR`.

Accessing information in a database view is illustrated in the constructor for `SportsDiaryEntry`:

```
self.timestamp  = r.col(1)
self.duration   = r.col(2)
self.distance   = r.col(3)
self.sport      = r.col(4)
self.comment    = r.col(5)
```

The `col` function of the `DB_view` type knows how to convert the database data types into Python types – see *Python for Series 60 Platform API Reference [2]* for details on how this is done. Since duration has been defined in the database table as having the type float, `self.duration` is also a float at the time when the constructor completes.

Figure 11 shows a sample sports diary entry.



Date	04/04/2005
Time	09:51:41
Duration	01:02:03
Distance	15
Sport	Running
Comment:	Felt ok

Figure 11: Sports diary

The `sql_add` method constructs a suitable SQL statement to insert a `SportsDiaryEntry` instance into the database.

```
def sql_add(self):
    sql = "INSERT INTO events (date, duration, distance, sport, comment)
VALUES (#%s#,%d,%d,%d,'%s')"%(
            e32db.format_time(self.timestamp),
            self.duration,
            self.distance,
            self.sport,
            self.comment)
    return unicode(sql)
```

Note: You must use single quotes (') around character strings in SQL statements.

The methods `get_form` and `set_from_form` deal with creating Series 60 forms. Forms are the most powerful UI elements, and they usually work well with database applications. However, this example concentrates only on formatting the data that the `form` uses.

```
def get_form(self):
    # Convert Unix timestamp into the form the form accepts.
    (yr, mo, da, h, m, s, wd, jd, ds) = \
         time.localtime(self.timestamp)
    m += 60*h # 60 minutes per hour
    s += 60*m # 60 seconds per minute
    result = [(u"Date", 'date', float(self.timestamp-s)),
              (u"Time", 'time', float(s)),
              (u"Duration", 'time', float(self.duration)),
              (u"Distance", 'number', int(self.distance))]
    if self.sport == None:
        result.append((u"Sport", 'combo', (self.sports, 0)))
    else:
        result.append((u"Sport", 'combo', (self.sports, self.sport)))
    result.append((u"Comment", 'text', self.comment))
    return result
```

Forms use lists as the data that they show, and after they return, their data can also be accessed as a list. The list consists of entries of type

```
u"  Label text", type, value
```

or, in case of selection from list (combo),

`u"Label text", 'combo',` (List of entries, initial choice).

For the `date` and `time` types, the value has to be a float. This can be ensured by casting the values. Note that the phone number field type is applied for `distance` to restrict the entry to numbers only. In this case, also a decimal point should be accepted, but the phone number field does not allow it and a compromise has to be made by treating `distance` as an integer. Therefore, it is not possible to enter decimal numbers, and `distance` is actually an integer. An alternative would be to use a text field and then parse it as a float, but entering numbers in a text field is harder.

The `sport` entry in the form gives a list of sports as alternatives, and uses 'Running' as a default if the sport has not been set to something else in this entry.

The `set_from_form` method is the inverse of `get_form`. It is used to parse the result after the users have returned from the form. The data is in the same form as in `get_form`.

```
def set_from_form(self, form):
    self.timestamp = form[0][2]+form[1][2]
    self.duration  = form[2][2]
    self.distance  = form[3][2]
    self.sport     = form[4][2][1]
    self.comment   = form[5][2]
```

The form makes all the data available, since some of its variants allow the users to modify, for instance, the labels. The first entry in the list form corresponds to the tuple

```
u"Date", 'date', <value>
```

The third element (index 2) is taken here since it contains the value.

All other cases are similar except for sport. The element in the list is now

```
    form[4] = u"Sport", 'combo', ([u"Running",…], <selection>)
```

Users' selection can be seen in Figure 12. The choice can be accessed by selecting the second element of the third element: `form[4][2][1]`.

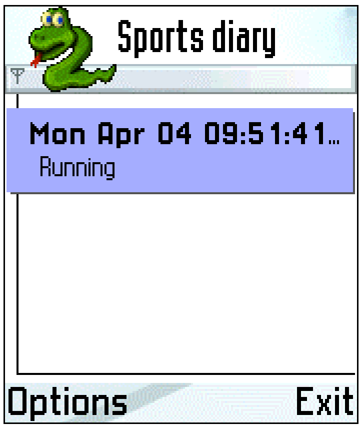


Figure 12: Users' selection

The `SportDiaryApp` represents a relatively standard program. The interesting parts concern adding and viewing entries. The implementation of the viewing functionality has been left as an exercise. The first entry is viewed here since it is simpler. Note that this view can only be accessed with the navigation key, not through the menu. The essential information is in the following method:

```
    def show_entry(self, entry):
        data = entry.get_form()
        flags = appuifw.FFormViewModeOnly
        f = appuifw.Form(data, flags)
        f.execute()
```

The entry is asked to get the suitable data for the form, as above. The flag this time is `FFormViewModeOnly`, since the users should not edit the entry (of course, a way for the users to edit the entries could also be added). The form is created and its `execute` method is called to make it visible. When the users select **Back**, the form closes.

It is recommended that when a new entry is added, the database and the display be updated:

```
    def handle_add(self):
        new_entry = SportsDiaryEntry()
        data = new_entry.get_form()
        flags = appuifw.FFormEditModeOnly
        f = appuifw.Form(data, flags)
        f.execute()
        new_entry.set_from_form(f)
        self.sports_diary.add(new_entry)
        self.lock.signal()
```

This time, the flag is `FFormEditModeOnly` that allows for editing the form. After the `execute` function returns, the form is updated with the new information that the users entered. Note that after `execute` returns, the form is no longer visible but it still exists and can be passed to the `set_from_form` method.

Now the `set_from_form` method is used to update the `SportsDiaryEntry` from the form. It is then added to the sports diary and the `main` function is told that it is time to update the display (a `Listbox`).

Finally, there is a little trick in `handle_delete`:

```
    def handle_delete(self):
        if self.entry_list:
            index = self.main_view.current()
        if appuifw.query(u"Delete entry?", 'query'):
            self.sports_diary.delete(self.entry_list[index])
        self.lock.signal()
```

Instead of blindly deleting the entry, the application asks the users for a confirmation. A query returns `True` or `False` depending on what the users selected.

10 Contacts and Calendar Databases

Calendar databases, todo entries, and todo lists can be accessed from the `calendar` module. Similarly, contact databases can be accessed from the `contacts` module. This chapter presents some examples on handling calendar appointments and contact entries.

For more examples, see *Appendix K*, *Contacts and Calendar Examples*. Also, see *Python for Series 60 Platform API Reference* [*2*].

The source code for the examples in this chapter and *Appendix K*, *Contacts and Calendar Examples* can be found in `test_contacts.py` and `test_calendar.py`.

10.1 Calendar Appointments

The following is an example of adding new, repeating calendar appointments into the calendar database:

To open the calendar database, do as follows:

```
import time
import calendar
week = 7*24*60*60
day = 24*60*60
hour = 60*60
now=time.time() # now it is 20.june.2005

db = calendar.open()
```

To create an appointment, do as follows:

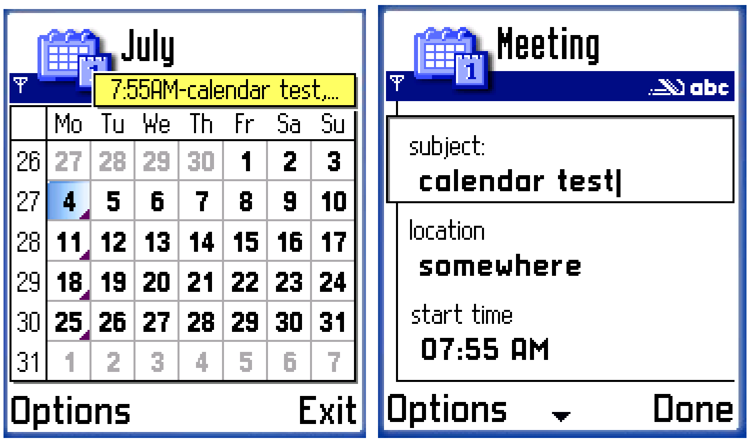
```
new_entry = db.add_appointment()
new_entry.set_time(now+2*week,now+2*week+hour)
new_entry.content='calendar test'
new_entry.location='somewhere'
```

To set the appointment to be repeated weekly for four weeks, do as follows:

```
repeat={"type":"weekly",
        "start":new_entry.start_time,
        "end":new_entry.start_time+4*week-day}
new_entry.set_repeat(repeat)

new_entry.commit()
```

Figure 13 shows how this looks in the native calendar application of your Nokia device.



	Mo	Tu	We	Th	Fr	Sa	Su
26	27	28	29	30	1	2	3
27	4	5	6	7	8	9	10
28	11	12	13	14	15	16	17
29	18	19	20	21	22	23	24
30	25	26	27	28	29	30	31
31	1	2	3	4	5	6	7

subject:
calendar test

location
somewhere

start time
07:55 AM

Figure 13: An example view of the Calendar application

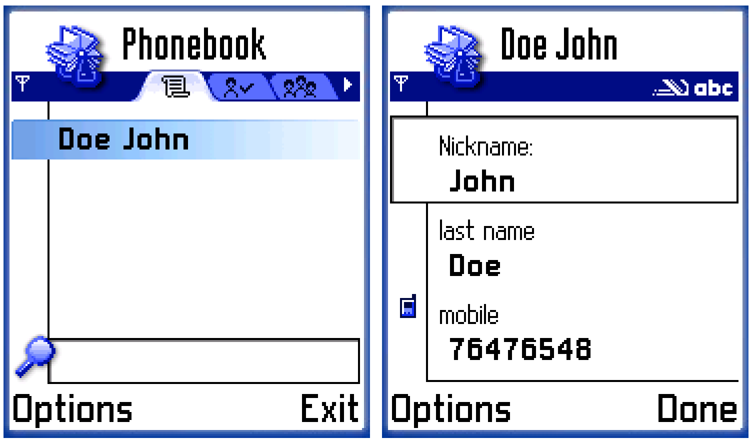
10.2 Contact Entries

The following is an example of adding a new contact entry:

```
import contacts

db = contacts.open()
contact = db.add_contact()
contact.add_field('first_name',value='John',label='Nickname')
contact.add_field('last_name','Doe')
contact.add_field('mobile_number','76476548')
contact.commit()
```

Figure 14 shows how this looks in the native Phonebook application of your Nokia device:



Doe John

Nickname:
John

last name
Doe

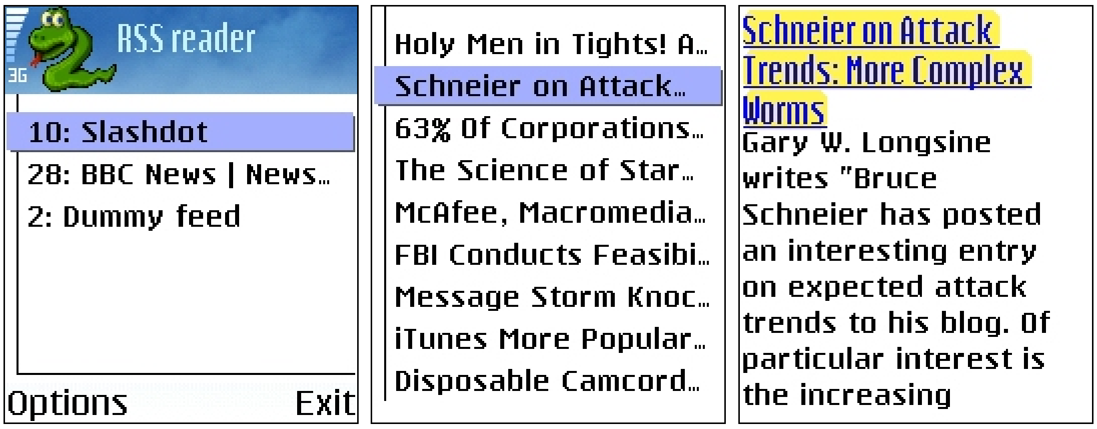
mobile
76476548

Figure 14: An example view of the Contacts application

For more examples, see *Appendix K, Contacts and Calendar Examples*. Also, see *Python for Series 60 Platform API Reference* [2].

11 Handling Key Bindings: RSS Reader

Figure 15 shows the RSS reader, which is a UI application that follows the structure introduced earlier (see Section *4.3*, *Application Skeleton*). RSS is a format for news feeds. The application uses key bindings for moving between the different views of the application. Moving right in the feed menu (first image of Figure 15) opens the articles in the selected feed in the article menu. Moving right from the article menu opens the article view (third image).


RSS reader

10: Slashdot
28: BBC News | News…
2: Dummy feed

Options Exit

Holy Men in Tights! A…
Schneier on Attack…
63% Of Corporations…
The Science of Star…
McAfee, Macromedia…
FBI Conducts Feasibi…
Message Storm Knoc…
iTunes More Popular…
Disposable Camcord…

Schneier on Attack Trends: More Complex Worms
Gary W. Longsine writes "Bruce Schneier has posted an interesting entry on expected attack trends to his blog. Of particular interest is the increasing

Figure 15: RSS reader with the screen mode set to **full** in the last two screens

Only a few parts of the source code are described here in detail. For the full source code, see *Appendix H, Source Code for RSS Reader*.

First of all, the program makes the application more responsive:

```
import thread
def import_modules():
    import simplefeedparser as feedparser
    import btconsole
thread.start_new_thread(import_modules,())
```

This imports modules in parallel in a separate thread, which works because all threads share a common namespace. The obvious benefit for the users is that the application does not stall while it is importing large libraries.

The key bindings are done as follows:

```
self.articlemenu=appuifw.Listbox([u''],self.handle_articlemenu_select)
self.articlemenu.bind(EKeyRightArrow,self.handle_articlemenu_select)
self.articlemenu.bind(EKeyLeftArrow,self.handle_exit)
```

The effect of pressing the right arrow (navigation key to the right) is the same as selecting an item, whereas pressing the left arrow is the same as pressing the right softkey.

Another slow operation is fetching the RSS feeds using GPRS. To save time, the application caches the articles in a DBM-type repository. This is done by opening a DBM store

```
cache=anydbm.open(u'c:\\rsscache','c')
```

and passing it to `CachingRSSFeed` constructor:

```
CachingRSSFeed(url='http://slashdot.org/index.rss',
               title='Slashdot',
               cache=cache)
```

All feeds are kept in the cache. The `CachingRSSFeed` registers a callback to invalidate the cache when there is more recent information available. The cache is updated in the permanent memory with

```
self.cache[self.url]=repr(self.feed)
```

in the `save` method of `CachingRSSFeed`.

To find out when a feed has updated itself, the `RSSFeed` class defines the `listen` method to allow all interested listeners to register a callback function. The `ReaderApp` registers interest in all feeds with

```
for k in self.feeds:
    k.listen(self.notify)
```

The `notify` method is in `ReaderApp` and refreshes the UI. The `ReaderApp` keeps track of its own state with the state attribute and manages transitions between states according to the state map.

For example, if the users decide to update a feed, the process is as follows:

1. Selection of **Update feed** from a menu calls the `handle_update` method in `ReaderApp`.
2. The `start_update` method on the feed in question is called.
3. The `start_update` method first checks if an update is already ongoing, and if not, creates a new thread that calls the internal `_update` method.
4. The `_update` method listens to the incoming information and parses it into intelligible form. Information keeps coming and is parsed as it arrives.
5. When all information has been parsed, the `_update` method calls the `notify_listeners` method that goes over the list of callbacks and makes sure they are called. Since the `notify` method of `ReaderApp` has been registered, that method gets called.
6. As before, the `notify` method signals the lock where the main thread is waiting in the `loop` method.
7. In the `loop` method, the main thread calls the `refresh` method before going back to wait on the lock.
8. Finally, the `refresh` method updates the display.

The screen mode setting is created by using a submenu. Giving a list item to the menu list, in the following format, creates the submenu:

```
(u'Screen mode', ((u'full', lambda:handle_screen('full')),
                  (u'large', lambda:handle_screen('large')),
                  (u'normal', lambda:handle_screen('normal')))),
```

where the function to set the screen mode is:

```
def handle_screen(mode):
    appuifw.app.screen = mode
```

Rich text appearance is set using `Text` object attributes as follows:

```
self.articleviewer.highlight_color = (255,240,80)  #set yellow higlight
self.articleviewer.style =
appuifw.STYLE_UNDERLINE|appuifw.HIGHLIGHT_ROUNDED
# set the text to be underlined and highlighted
# with rounded style highlight
self.articleviewer.font = 'title'
#set the font to be the same as used in application titles in device
self.articleviewer.color = (0,0,255)   #sets the font color to blue
```

12 Real-Time Graphics Support and Key Event Handling: ball.py

The current Python for Series 60 distribution includes two objects that can act as a target for graphics drawing operations: `Canvas` and `Image`. `Image` represents an in-memory drawing surface, whereas `Canvas` represents an actual drawing area on the screen. `Image` objects are often useful as background buffers and temporary drawing surfaces.

Only a few parts of the source code are described here in detail. For the full source code, see *Appendix L, Source Code for Ball.*

The `ball.py` example is a typical full-screen graphical application. At the beginning of the program the screen layout is switched to the **full** mode and a `Canvas` object is created. The constructor parameters of `Canvas` are callbacks for key and redraw events.

```
appuifw.app.screen='full'
img=None
def handle_redraw(rect):
    if img:
        canvas.blit(img)
appuifw.app.body=canvas=appuifw.Canvas(
    event_callback=keyboard.handle_event,
    redraw_callback=handle_redraw)
img=Image.new(canvas.size)
```

An exit key handler that exits the application gracefully is useful in most nontrivial Python for Series 60 applications:

```
running=1
def quit():
    global running
    running=0
appuifw.app.exit_key_handler=quit
```

12.1 Drawing and Redrawing

Each time you go through the application main loop, you need to clear the screen and draw all objects on to the screen again. This could be done directly in `Canvas`, but that would lead to flickering, since while the drawing is taking place the user would see a partially completed picture on the screen. Therefore, you should use an `Image` object as a temporary buffer. The objects are first drawn onto this `Image` object and its contents are then transferred to the screen with one blit operation. This technique, known as double buffering, removes the flickering since half-finished drawings are never seen on the screen.

This same `Image` object is also useful when redrawing the `Canvas` contents after something has drawn over the `Canvas`. Whenever the UI framework detects that something has drawn onto the space occupied by the `Canvas`, the `redraw_callback` given as the `Canvas` constructor parameter is called. Redrawing is very simple when you have a backup `Image`:

```
def handle_redraw(rect):
    if img:
        canvas.blit(img)
```

Giving this redraw method to the `Canvas` is not strictly necessary here, since in this application the screen is being completely redrawn frequently in any case.

12.2 Key Event Handling

The `Canvas` constructor parameter `event_callback` provides access to raw key events from the keyboard. Whenever a key event occurs, the callback is called with the event as the parameter.

For this application, you need to be able to access the following information:

- Is a specific key currently pressed down? (For arrow keys)
- Has a specific key been pressed after the last time through the loop? (For the screen shot key)

To obtain this information, create a helper class that will keep track of which keys are currently down, and how many times each key has been pressed:

```
class Keyboard(object):
    def __init__(self,onevent=lambda:None):
        self._keyboard_state={}
        self._downs={}
        self._onevent=onevent
    def handle_event(self,event):
        if event['type'] == appuifw.EEventKeyDown:
            code=event['scancode']
            if not self.is_down(code):
                self._downs[code]=self._downs.get(code,0)+1
            self._keyboard_state[code]=1
        elif event['type'] == appuifw.EEventKeyUp:
            self._keyboard_state[event['scancode']]=0
        self._onevent()
    def is_down(self,scancode):
        return self._keyboard_state.get(scancode,0)
    def pressed(self,scancode):
        if self._downs.get(scancode,0):
            self._downs[scancode]-=1
            return True
        return False
keyboard=Keyboard()
```

With this helper object, you can check, for example, whether the left arrow key is down by using `keyboard.is_down(EScancodeLeftArrow)`, or whether the **Hash** key has been pressed since the last time of checking by using `keyboard.pressed(EScancodeHash)`. The `pressed` method returns `True` as exactly as many times as the key has been pressed.

12.3 Main Loop

Clear the backup buffer with black and draw each object onto it:

```
img.clear(0)
img.text((0,14),u'Use arrows to move ball',0xffffff)
img.point((location[0]+blobsize/2,location[1]+blobsize/2),
           0x00ff00,width=blobsize)
handle_redraw(())
```

Draw the backup buffer onto the screen:

```
handle_redraw(())
```

Handle any pending events. Possible key callbacks also get called at this point:

```
e32.ao_yield()
```

If the **Hash** key has been pressed, save a screen shot to the memory card in PNG format. For simplicity, the notification text is drawn directly on the `Canvas` instead of the backup bitmap:

```
if keyboard.pressed(EScancodeHash):
    filename=u'e:\\screenshot.png'
    canvas.text((0,32),u'Saving screenshot to:',fill=0xffff00)
    canvas.text((0,48),filename,fill=0xffff00)
    img.save(filename)
```

13 Python Execution Environment: default.py and Others

A script called `default.py` script implements most of the visible functionality of the Python execution environment. The script is executed when the **Python** icon is selected on the main menu. When users select **Python**, a little native (C++) `launchpad` application starts. It loads the Python interpreter library, reads `default.py`, and runs it. If you want to modify the default behavior, edit the `default.py` file. (The stand-alone Python applications are implemented in a similar manner. A tool is available for creating stand-alone Python applications installable as SIS packages; see Chapter *14*, *Making Stand-Alone Applications from Python Scripts*.)

For the source code of the standard `default.py` file, see *Appendix I*, *Source Code for default.py*.

The `init_options_menu` function sets the application menu contents. The `query_and_exec` function creates a list of scripts in the script directory. To find out where it is running, it uses `app.full_name` to find the full path name to the running application – that is, the location of `default.py`. The path part is in the `this_dir` variable to be used by the `query_and_exec` function. Scripts are in `this_dir` or in its subdirectory `my`. The script file is executed with the standard `execfile` command.

The default display is the `series60_console` module that defines `Console` class. It creates a `Text` UI control and defines `clear`, `write`, `writelines`, `flush`, and `readline` methods. It then redirects `stdin`, `stdout`, and `stderr` to itself.

The benefit of this arrangement is that when a Python script for a console is running, it finds the standard input and output redirected to a console that knows how to deal with them. Therefore, if the script `hello.py` that contains the line

```
print "Hello"
```

`is run, it prints the output to stdout, which is then redirected to the write method of Console`. In this way the print statement actually applies a UI control.

As discussed earlier, only the UI thread should do output to UI controls. If the print statement is called from a non-UI thread, the output goes to a buffer. Otherwise, it is appended to the end of the buffer and the buffer is output.

If the application redirects `stdout`, the output may never reach the `series60_console`.

The interactive console that is launched with `exec_interactive` uses `series60_console` as a basis of its implementation. It adds menu items and binds the **Enter** key to a command. The actual work in interactive console is done in the Python native `code.interact` function.

14 Making Stand-Alone Applications from Python Scripts

`py2sis` is a utility for packaging a Python script as a SIS file to be installed in the Symbian Series 60 1st and 2nd Edition devices. `py2sis` comes with the Python for Series 60 SDK installation package. Python for Series 60 needs to be installed on the target device since the stand-alone applications depend on it. Before packaging your script it is a standard procedure to verify that your script does not contain defects, for example by running it successfully with the Python application.

Tip: For more useful information on `py2sis`, search the Python for Series 60 developer discussion board with the key term "py2sis" [*5*].

Use the command line utility in the following way:

```
py2sis <src> [sisfile] [--uid=0x12345678] [--appname=myapp] [--presdk20]
[--leavetemp]
```

giving the path to the script or directory as `<src>` parameter. If you are packaging a whole directory, the directory must contain a file named `default.py` which will be used as the main script. If the directory from where `py2sis` is invoked contains a file named `default.py`, this file will be included in the created package instead of the file given as the command line parameter. This directory can also contain other files which your application needs, for example WAV files, PNG files, and an AIF file. An AIF file contains icons for an application. These icons show in the device **Menu** and in the status pane of your application. For generating AIF files, please refer to, for example, *Series 60 SDK Help* documentation [*3*] using the key terms *"How to compile an aif file"*and *"Introduction to AIF Builder"*.

Note: In Series 60 2nd Edition FP3 and further releases, using SVG icons with `py2sis` is not supported. However, it is possible to manually edit the created `.pkg` file to include resolution-dependent icons. For more information on this, search Series 60 SDK 2nd Edition FP3 Help documentation with the key term *"Package file"*.

`py2sis` uses the command line tools from the Symbian SDK, so the SDK needs to be installed and properly configured. This means that the `makesis` and `uidcrc` utilities need to be in your system path. This can be verified by running `makesis` and `uidcrc` before invoking `py2sis`.

By default, the SIS file is created in the current working directory, but optionally you can specify the path where you want to save the resulting SIS with the `sisfile` parameter.

Example: `py2sis myscript.py c:\mysis.sis`

All Symbian applications need to have an UID, which you can provide from the command line using the `--uid` switch. The UID can also be embedded in the main script by including the line:

```
# SYMBIAN_UID = 0x01234567
```

The UIDs must be assigned properly, and during development time you can use temporary UIDs from the range of `0x01000000` to `0x0fffffff`. For more information on UIDs, see *Series 60 SDK Help* documentation [*3*] using the key terms *"How to use UIDs"*.

The name of the application is taken from the source name, but the name can also be specified using the `--appname` switch.

If you are packaging a SIS file for a Series 60 1st Edition device, you must use switch `--presdk20`.

For investigating what was packaged into a SIS file, use switch `--leavetemp`[1]; this will not delete the temporary directory `temp/` used in the packaging process.

[1] See also the `unsis` tool on the Symbian OS Tools Web page for unpacking the created SIS packages [4].

15 Porting Python Applications for PC to Series 60

Many Python applications made for PCs work without modification on Python for Series 60. The most noticeable differences are in the UI and the availability of other capabilities. Here are some guidelines if the application does not work as such:

- If the application depends on a Python module that is not installed on the phone, see if it is possible to add the module. Modules often depend on other modules. The extension of the module on the PC tells its type: `.py` extension means that the extension is written in Python, and `.pyd` means that it is written in C.
- If the application depends on a Python extension written in C, the only alternative is to port the module to Symbian. For some instructions on this, see *Python for Series 60 Platform API Reference [2]*. Notice, however, that this requires familiarity with Symbian C++ programming.

While applications that use only the console functionality (writing and reading text) work as such on Series 60, the screen may not be updated correctly when the application is computing without doing OS calls. Running the computation in a separate thread or occasionally calling `e32.ao_yield` in the main thread will give the system a chance to process UI events and prevent the UI from freezing.

See also Chapter *16*, *Porting a Simple Extension to Series 60*.

16 Porting a Simple Extension to Series 60

Python for Series 60 supports the Python/C API for writing your own extension modules in C or C++. There are some differences between Symbian OS and more commonly used operating systems that require you to make a few changes to your extension module before it will work with Python for Series 60.

This chapter guides you through the necessary steps for a simple extension `elemlist`, originally written by Alex Martelli for the *Python Cookbook*. The extension implements a new type known as a `cons cell`, which is similar to a two-element tuple and to the cons cells used in Lisp. The full source code is listed in *Appendix J*, *Source Code for Example Extension*. For some more tips, see *Appendices B* and *C* of the *Python for Series 60 Platform API Reference [2]*.

16.1 Required Modifications to the Example Extension

Symbian OS does not support writable static data in DLLs, which means that all static variables must be either converted into constants or moved to memory allocated from the heap. A common case where this is needed is when you define a new type in your extension module and need to allocate a type object for it. Python extensions for other platforms often simply define the type object as a static struct and use it directly, but that approach does not work in Symbian.

In the example, `cons_type` has been converted into a constant `const_type_template`. In the module initialization function, a new type object is allocated and the contents of `const_type_template` are copied to it. A reference to the newly allocated type object is stored in the module dictionary using an extension to the Python/C API, `SPyAddGlobalString`. To make it easier to access this type object, a macro `cons_type` is defined that looks up the type object using `SPyGetGlobalString`.

Depending on your SDK, you might also need to define the environment variable `EPOCROOT` to `EPOCROOT=\`.

Since most Symbian APIs are based on C++, you typically need to compile all modules that access Symbian APIs with a C++ compiler. This was also done in this example.

To make the extension compile, you need the build files `bld.inf` and `elemlist.mmp`. You should be able to use the build files from this example, with minor modifications, to compile your own extensions. For complete details on these build files, see *Series 60 SDK Help* documentation [3].

16.2 Installing the Example

1. Unpack the example code to the drive where you installed your Series 60 SDK, to a directory that is on the same level as the `epoc32` directory of the SDK. For example, if your `epoc32` directory is `c:\symbian\7.0s\series60_v20\epoc32` unpack the code in `c:\symbian\7.0s\series60_v20\example`.
2. Make sure you have defined a virtual drive that points to the directory that contains the `epoc32` directory. For example, if your `epoc32` directory is in `c:\symbian\7.0s\series60_v20`, you can do this with the following command:
   ```
   subst s: c:\symbian\7.0s\series60_v20
   ```

16.3 Compiling the Example

A script file `build_all.cmd` that does all of the necessary steps — and some extra cleanup, for certainty — has been provided for convenience. You can either use that or perform the build manually using these step-by-step instructions.

To perform the build manually:

1. Go to the example directory. Enter:
   ```
   bldmake bldfiles
   ```
2. To build the extension:
   - For the phone, enter:
     ```
     abld build armi urel
     abld freeze
     abld build armi urel
     ```
   - For the emulator environment, enter:
     ```
     abld build wins udeb
     abld freeze
     abld build wins udeb
     ```

Note: The `freeze` step only needs to be performed once. After changing the code, only one `abld build armi urel`, or `abld build wins udeb`, will rebuild the code properly.

3. To find the built module:
   - For the phone build, you should find the built module in:
     `(path to your SDK)\epoc32\release\armi\urel\elemlist.pyd`
     Transfer the `elemlist.pyd` file to your phone in the `\system\libs directory`.
   - For the emulator build, you should find the built module in:
     `(path to your SDK)\epoc32\release\wins\udeb\z\system\libs\elemlist.pyd`
     It should be available to the emulator right away.

16.4 Running the Example

Start the interpreter and try the following code:
```
from elemlist import *
cell=cons(1,2)
car(cell)
cdr(cell)
```

The results of the last two commands should be 1 and 2 respectively.

17 Terms and Abbreviations

The following list defines the terms and abbreviations used in this document:

AIF	An AIF file contains the caption, icon, capabilities, and MIME priority support information for an application. Has the file extension .aif.[2]
API	Application Programming Interface
Bluetooth	Bluetooth is a technology for wireless communication between devices that is based on a low-cost short-range radio link.
DBM	A set of database routines that uses extensible hashing. The `dbm` module provides an interface to the Unix `(n)dbm` library.
Dialog	A temporary user interface window for presenting context-specific information to the user, or prompting for information in a specific context.
Discovery	Discovery is a process where Bluetooth wireless technology finds other nearby Bluetooth devices and their advertised services.
DLL	Dynamic link library
Navigation key	A joystick-like directional input control key.
RSS	A web content distribution and republication protocol.
SMS; Short Message System (within GSM)	SMS is a service for sending messages of up to 160 characters (224 characters if using a 5-bit mode) to mobile phones that use GSM communication.
SIS	Symbian installation file, produced by the installation file generator or the SIS file creator. Python scripts can be packaged into installation files using the `py2sis` tool.
Softkey	Softkey is a key that does not have a fixed function or a function label printed on it. On a phone, selection keys reside below or above the screen, and derive their meaning from what is presently on the screen.
SQL	Structured Query Language
UI	User Interface
UI control	UI control is a GUI component supported by Series 60 that enables user interaction and represents properties or operations of an object.
UID; Unique Identifier	A UID is a globally unique 32-bit number used in a compound identifier to uniquely identify an object, file type, etc. When users refer to "UID" they often mean UID3, the identifier for a particular program.

[2] Description based on information found in Series 60 SDK documentation [*3*].

18 References

1. Getting Started with Python for Series 60 Platform

   http://www.forum.nokia.com/

2. Python for Series 60 Platform API Reference

   http://www.forum.nokia.com/

3. Series 60 SDK Help documentation

4. Symbian OS Tools
   http://www.symbian.com/developer/downloads/tools.html#unsis

5. Python for Series 60 developer discussion board
   http://discussion.forum.nokia.com/

Appendix A Source Code for Weather Maps

```
# Copyright (c) 2004 Nokia
# Programming example -- see license agreement for additional rights
# Simple GUI example

import socket
import urllib

import e32
import appuifw

# List of triplets "Name", "URL", "extension"
choices=[(u"US Graphical Forecast",
"http://weather.gov/forecasts/graphical/images/thumbnail/Thumbnail_Wx4_c
onus.png", "png"),
         (u"US Radar Image",
"http://weather.gov/mdl/radar/rcm1pix_b.gif", "gif"),
         (u"US Satellite Image",
"http://weather.gov/satellite_images/national.jpg", "jpg") ]
tempfile_without_extension = "c:\\weather"

old_title = appuifw.app.title
appuifw.app.title = u"Weather forecast"

L = [ x[0] for x in choices ]
index = appuifw.popup_menu(L, u"Select picture")

if index is not None:
    url = choices[index][1]
    ext = choices[index][2]
    tempfile = tempfile_without_extension + "." + ext

    try:
        print "Retrieving information..."
        urllib.urlretrieve(url, tempfile)
        lock=e32.Ao_lock()
        content_handler = appuifw.Content_handler(lock.signal)
        content_handler.open(tempfile)
        # Wait for the user to exit the image viewer.
        lock.wait()
        print "Image viewer finished."
    except IOError:
        print "Could not fetch the image."
    except:
        print "Could not open data received."

appuifw.app.title = old_title
```

Appendix B Source Code for Weather Information

```
# Copyright (c) 2004 Nokia
# Programming example -- see license agreement for additional rights
# Simple GUI example 2

import socket
import urllib

import e32
import appuifw

choices =[(u"Los Angeles Intl Airport", "KLAX"),
          (u"Dallas/Fort Forth", "KDFW"),
          (u"New York/John F. Kennedy", "KJFK")]
choices_labels = [x[0] for x in choices]

weather_url_base = "http://weather.gov/data/current_obs/"
tempfile = "c:\\weather.xml"

def find_value(text, tag):
    "Find the value between <tag> and </tag> in text. Always returns a
string"
    begin_tag = "<" + tag + ">"
    begin = text.find(begin_tag)
    end = text.find("</" + tag + ">")
    if begin == -1 or end == -1:
        return ""
    begin += len(begin_tag)
    return text[begin:end]

def handle_selection():
    index = lb.current()
    code = choices[index][1]
    weather_url = weather_url_base + code + ".xml"
    lb.set_list([u"Please wait..."])
    appuifw.note(u"Fetching "+ weather_url, 'info')
    try:
        urllib.urlretrieve(weather_url, tempfile)
        f = open(tempfile, 'r')
        weatherinfo = f.read()
        f.close()
        weather = find_value(weatherinfo, "weather")
        temperature_string = find_value(weatherinfo,
                                        "temperature_string")
        appuifw.popup_menu([(u"Weather", unicode(weather)),
                            (u"Temperature",
                             unicode(temperature_string))],
                           unicode(code))
    except IOError:
        appuifw.note(u"Connection error to server", 'error')
    except:
        appuifw.note(u"Could not fetch information", 'error')
    lb.set_list(choices_labels)

def handle_add():
    pass

def handle_delete():
    pass

def exit_key_handler():
    app_lock.signal()

lb = appuifw.Listbox(choices_labels, handle_selection)

old_title = appuifw.app.title
```

```
appuifw.app.title = u"Weather report"
appuifw.app.body = lb
appuifw.app.menu = [(u"Add new item", handle_add),
                    (u"Delete item", handle_delete)]
appuifw.app.exit_key_handler = exit_key_handler

app_lock = e32.Ao_lock()
app_lock.wait()

appuifw.app.title = old_title
```

Appendix C Source Code for Application Skeleton

```
# Copyright (c) Nokia 2004
# Programming example -- see license agreement for additional rights
# Advanced GUI example

# This nonfunctional sample code is based on a simple application for
# accessing a to-do list. The details of that particular application
# have been edited out.

import e32
import appuifw

from MyDataAccess import MyDataAccess

e32.ao_yield()

def format(item):
    # Format the item as a short unicode string.
    return u"" # add your own code here

class MyApp:
    def __init__(self):
        self.lock = e32.Ao_lock()

        self.old_title = appuifw.app.title
        appuifw.app.title = u"My Application"

        self.exit_flag = False
        appuifw.app.exit_key_handler = self.abort

        self.data = []
        appuifw.app.body = appuifw.Listbox([u"Loading..."],
                                           self.handle_modify)

        self.menu_add = (u"Add", self.handle_add)
        self.menu_del = (u"Delete", self.handle_delete)
        appuifw.app.menu = []
               # First call to refresh() will fill in the menu.

    def connect(self, host):
        self.db = MyDataAccess(host)
        self.db.listen(self.notify)
               # Set up callback for change notifications.

    def loop(self):
        try:
            self.lock.wait()
            while not self.exit_flag:
                self.refresh()
                self.lock.wait()
        finally:
            self.db.close()

    def close(self):
        appuifw.app.menu = []
        appuifw.app.body = None
        appuifw.app.exit_key_handler = None
        appuifw.app.title = self.old_title

    def abort(self):
        # Exit-key handler.
        self.exit_flag = True
        self.lock.signal()

    def notify(self, in_sync):
        # Handler for database change notifications.
```

```
            if in_sync:
                self.lock.signal()

        def refresh(self):
            # Note selected item.
            current_item = self.get_current_item()

            # Get updated data.
            self.data = self.db.get_data()

            if not self.data:
                content = [u"(Empty)"]
            else:
                content = [format(item) for item in self.data]

            if current_item in self.data:
                # Update the displayed data,
                # retaining the previous selection.
                index = self.data.index(current_item)
                appuifw.app.body.set_list(content, index)
            else:
                # Previously selected item is no longer present, so allow
                # the selection to be reset.
                appuifw.app.body.set_list(content)

            if not self.data:
                appuifw.app.menu = [self.menu_add]
            else:
                appuifw.app.menu = [self.menu_add, self.menu_del]

        def handle_modify(self):
            item = self.get_current_item()
            if item is not None:
                # Display data in Form for user to edit.
                # Save modified record in database.
                pass                        # omitted

        def handle_add(self):
            new_item = self.edit_item(ToDoItem())
            if new_item is not None:
                # User enters new data into Form.
                # Save new record in database.
                pass                        # omitted

        def handle_delete(self):
            item = self.get_current_item()
            if item is not None:
                # Remove record from database.
                pass                        # omitted

        def get_current_item(self):
            # Return currently selected item, or None if the list is empty.
            if not self.data:
                return None
            else:
                current = appuifw.app.body.current()
                return self.data[current]

    def main():
        app = MyApp()
        try:
            hosts = [u"some.foo.com", u"other.foo.com"]
            i = appuifw.popup_menu(hosts, u"Select server:")
            if i is not None:
                app.connect(hosts[i])
                app.loop()
        finally:
            app.close()
```

```
if __name__ == "__main__":
    main()
```

Appendix D Source Code for SMS Sending

```
# Copyright (c) 2004 Nokia
# Programming example -- see license agreement for additional rights
# SMS sending example application

import appuifw
import e32
# import messaging

old_title = appuifw.app.title
appuifw.app.title = u"SMS sending"


class NumbersView:
    def __init__(self, SMS_multiviewApp):
        self.SMS_multiviewApp = SMS_multiviewApp
        self.dict = [(u"Jim", "55512345"), (u"Jane", "55567890")]
        self.names = [item[0] for item in self.dict]
        self.numbers = [item[1] for item in self.dict]

        self.numbers_list = appuifw.Listbox(self.names,
self.handle_select)
        self.index = None
        appuifw.app.body = self.numbers_list

    def activate(self):
        appuifw.app.body = self.numbers_list
        appuifw.app.menu = [(u"Select", self.handle_select)]

    def handle_select(self):
        n = self.get_name()
        appuifw.note(u"Selected: "+ n, 'info')

    def get_current(self):
        return self.numbers_list.current()

    def get_name(self):
        i = self.get_current()
        return self.names[i]

    def get_number(self):
        i = self.get_current()
        return self.numbers[i]


class ChoiceView:
    def __init__(self, SMS_multiviewApp):
        self.SMS_multiviewApp = SMS_multiviewApp
        self.texts = [u"I am late",
                      u"What is for dinner?",
                      u"Do you need anything from the supermarket?",
                      u"How about a round of golf after work?"]
        self.listbox = appuifw.Listbox(self.texts, self.handle_select)

    def activate(self):
        appuifw.app.body = self.listbox
        appuifw.app.menu = [(u"Select", self.handle_select),
                            (u"Send", self.handle_send)]

    def handle_select(self):
        i = self.listbox.current()
        appuifw.note(u"Selected: " + self.get_text(),'info')

    def handle_send(self):
        appuifw.app.activate_tab(3)
        self.SMS_multiviewApp.handle_tab(3)
    def get_text(self):
        return self.texts[self.listbox.current()]


class TextView:
    def __init__(self, SMS_multiviewApp):
        self.SMS_multiviewApp = SMS_multiviewApp
        self.view_text = appuifw.Text()

    def activate(self):
        t = self.SMS_multiviewApp.get_text()
        self.view_text.set(t)
        appuifw.app.body = self.view_text
        appuifw.app.menu = [(u"Send", self.handle_send)]
        self.view_text.focus = True

    def handle_send(self):
        appuifw.app.activate_tab(3)
        self.SMS_multiviewApp.handle_tab(3)


class SendView:
    def __init__(self, SMS_multiviewApp):
        self.SMS_multiviewApp = SMS_multiviewApp
        self.log_text = appuifw.Text()
        self.log_contents = u""

    def activate(self):
        self.log_text.set(self.log_contents)
        appuifw.app.body = self.log_text
        appuifw.app.menu = []
        nbr = self.SMS_multiviewApp.get_number()
        txt = self.SMS_multiviewApp.get_text()
        nam = self.SMS_multiviewApp.get_name()
        if appuifw.query(u"Send message to " + nam + "?", 'query'):
            t = u"Sent " + txt + " to " + nbr + " (" + nam + ")\n"
            self.log_contents += t
            self.log_text.add(t)
            # messaging.sms_send(nbr, txt)


class SMS_multiviewApp:
    def __init__(self):
        self.lock = e32.Ao_lock()
        appuifw.app.exit_key_handler = self.exit_key_handler

        self.n_view = NumbersView(self)
        self.c_view = ChoiceView(self)
        self.t_view = TextView(self)
        self.s_view = SendView(self)
        self.views = [self.n_view, self.c_view, self.t_view,
self.s_view]
        appuifw.app.set_tabs([u"Numbers", u"Choice", u"Text", u"Send"],
                             self.handle_tab)

    def run(self):
        self.handle_tab(0)
        self.lock.wait()
        self.close()

    def get_name(self):
        return self.n_view.get_name()

    def get_number(self):
        return self.n_view.get_number()

    def get_text(self):
        return self.c_view.get_text()
    def handle_tab(self, index):
        self.views[index].activate()

    def exit_key_handler(self):
        self.lock.signal()

    def close(self):
        appuifw.app.exit_key_handler = None
        appuifw.app.set_tabs([u"Back to normal"], lambda x: None)
        del self.t_view
        del self.s_view

myApp = SMS_multiviewApp()
myApp.run()

appuifw.app.title = old_title
appuifw.menu = None
```

Appendix E Source Code for File Browser

```
#
# filebrowser.py
#
# A very simple file browser script to demonstrate the power of Python
# on Series 60.
#
# Copyright (c) 2004 Nokia. All rights reserved.
#

import os
import appuifw
import e32
import dir_iter

class Filebrowser:
    def __init__(self):
        self.script_lock = e32.Ao_lock()
        self.dir_stack = []
        self.current_dir = dir_iter.Directory_iter(e32.drive_list())

    def run(self):
        from key_codes import EKeyLeftArrow
        entries = self.current_dir.list_repr()
        if not self.current_dir.at_root:
            entries.insert(0, (u"..", u""))
        self.lb = appuifw.Listbox(entries, self.lbox_observe)
        self.lb.bind(EKeyLeftArrow, lambda: self.lbox_observe(0))
        old_title = appuifw.app.title
        self.refresh()
        self.script_lock.wait()
        appuifw.app.title = old_title
        appuifw.app.body = None
        self.lb = None

    def refresh(self):
        appuifw.app.title = u"File browser"
        appuifw.app.menu = []
        appuifw.app.exit_key_handler = self.exit_key_handler
        appuifw.app.body = self.lb

    def do_exit(self):
        self.exit_key_handler()

    def exit_key_handler(self):
        appuifw.app.exit_key_handler = None
        self.script_lock.signal()

    def lbox_observe(self, ind = None):
        if not ind == None:
            index = ind
        else:
            index = self.lb.current()
        focused_item = 0

        if self.current_dir.at_root:
            self.dir_stack.append(index)
            self.current_dir.add(index)
        elif index == 0:                              # ".." selected
            focused_item = self.dir_stack.pop()
            self.current_dir.pop()
        elif os.path.isdir(self.current_dir.entry(index-1)):
            self.dir_stack.append(index)
            self.current_dir.add(index-1)
        else:
            item = self.current_dir.entry(index-1)
            if os.path.splitext(item)[1] == '.py':
                i = appuifw.popup_menu([u"execfile()", u"Delete"])
            else:
                i = appuifw.popup_menu([u"Open", u"Delete"])
            if i == 0:
                if os.path.splitext(item)[1].lower() == u'.py':
                    execfile(item, globals())
                    self.refresh()
                    #appuifw.Content_handler().open_standalone(item)
                else:
                    try:
                        appuifw.Content_handler().open(item)
                    except:
                        import sys
                        type, value = sys.exc_info() [:2]
                        appuifw.note(unicode(str(type)+'\n'+str(value)),
                                     "info")
                return
            elif i == 1:
                os.remove(item)
                focused_item = index - 1

        entries = self.current_dir.list_repr()
        if not self.current_dir.at_root:
            entries.insert(0, (u"..", u""))
        self.lb.set_list(entries, focused_item)

if __name__ == '__main__':
    Filebrowser().run()
```

Appendix F Source Code for Bluetooth Sockets

```
# Copyright (c) 2004 Nokia
# Programming example -- see license agreement for additional rights
# A simple interactive console over Bluetooth wireless technology.

import socket

class socket_stdio:
    def __init__(self,sock):
        self.socket=sock
    def read(self,n=1):
        return self.socket.recv(n)
    def write(self,str):
        return self.socket.send(str.replace('\n','\r\n'))
    def readline(self,n=None):
        buffer=[]
        while 1:
            ch=self.read(1)
            if ch == '\n' or ch == '\r':   # return
                buffer.append('\n')
                self.write('\n')
                break
            if ch == '\177' or ch == '\010': # backspace
                self.write('\010 \010')
                     # erase character from the screen
                del buffer[-1:] # and from the buffer
            else:
                self.write(ch)
                buffer.append(ch)
            if n and len(buffer)>=n:
                break
        return ''.join(buffer)
    def raw_input(self,prompt=""):
        self.write(prompt)
        return self.readline()
    def flush(self):
        pass

sock=socket.socket(socket.AF_BT,socket.SOCK_STREAM)
# For quicker startup, enter here the address and port to connect to.
target='' #('00:20:e0:76:c3:52',1)
if not target:
    address,services=socket.bt_discover()
    print "Discovered: %s, %s"%(address,services)
    if len(services)>1:
        import appuifw
        choices=services.keys()
        choices.sort()
        choice=appuifw.popup_menu(
            [unicode(services[x])+": "+x for x in choices],
            u'Choose port:')
        target=(address,services[choices[choice]])
    else:
        target=(address,services.values()[0])
print "Connecting to "+str(target)
sock.connect(target)
socketio=socket_stdio(sock)
realio=(sys.stdout,sys.stdin,sys.stderr)
(sys.stdout,sys.stdin,sys.stderr)=(socketio,socketio,socketio)
import code
try:
  code.interact()
finally:
  (sys.stdout,sys.stdin,sys.stderr)=realio
  sock.close()
```

Appendix G Source Code for Sports Diary

```
# Copyright (c) 2004 Nokia
# Programming example -- see license agreement for additional rights
# Database example application: a sports diary.

import time

import e32
import e32db
import appuifw

class SportsDiary:
    def __init__(self, db_name):
        try:
            self.native_db = e32db.Dbms()
            self.native_db.open(db_name)
        except:
            self.native_db.create(db_name)
            self.native_db.open(db_name)
            self.native_db.execute(SportsDiaryEntry.sql_create)

    def get_all_entries(self):
        dbv = e32db.Db_view()
        dbv.prepare(self.native_db,
                    u"SELECT * from events ORDER BY date DESC")
        dbv.first_line()
        results = []
        for i in range(dbv.count_line()):
            dbv.get_line()
            e = SportsDiaryEntry(dbv)
            results.append(e)
            dbv.next_line()
        return results

    def add(self, e):
        self.native_db.execute(e.sql_add())

    def delete(self, e):
        self.native_db.execute(e.sql_delete())

    def close(self):
        self.native_db.close()

class SportsDiaryEntry:
    sports = [u"Running", u"Skating", u"Biking", u"Skiing", u"Swimming"]
    sql_create = u"CREATE TABLE events (date TIMESTAMP, duration FLOAT,
distance FLOAT, sport INTEGER, comment VARCHAR)"

    # Initialize with a row from Sport_diary_db
    def __init__(self, r=None):
        if r:
            self.timestamp  = r.col(1)
            self.duration   = r.col(2)
            self.distance   = r.col(3)
            self.sport      = r.col(4)
            self.comment    = r.col(5)
        else:
            self.timestamp  = time.time()
            self.duration   = 0.0
            self.distance   = 0.0
            self.sport      = None
            self.comment    = u""

    def sql_add(self):
        sql = "INSERT INTO events (date, duration, distance, sport,
comment) VALUES (#%s#,%d,%d,%d,'%s')"%(
            e32db.format_time(self.timestamp),
            self.duration,
            self.distance,
            self.sport,
            self.comment)
        return unicode(sql)

    def sql_delete(self):
        sql  = "DELETE FROM events WHERE date=#%s#"%\
              e32db.format_time(self.timestamp)
        return unicode(sql)

    def unixtime(self):
        return self.timestamp

    def get_sport_text(self):
        return self.sports[self.sport]

    def get_form(self):
        # Convert Unix timestamp into the form the form accepts.
        (yr, mo, da, h, m, s, wd, jd, ds) = \
             time.localtime(self.timestamp)
        m += 60*h # 60 minutes per hour
        s += 60*m # 60 seconds per minute
        result = [(u"Date", 'date', float(self.timestamp-s)),
                  (u"Time", 'time', float(s)),
                  (u"Duration", 'time', float(self.duration)),
                  (u"Distance", 'number', int(self.distance))]
        if self.sport == None:
            result.append((u"Sport", 'combo', (self.sports, 0)))
        else:
            result.append((u"Sport", 'combo', (self.sports,
                                               self.sport)))
        result.append((u"Comment", 'text', self.comment))
        return result

    def set_from_form(self, form):
        self.timestamp = form[0][2]+form[1][2]
        self.duration  = form[2][2]
        self.distance  = form[3][2]
        self.sport     = form[4][2][1]
        self.comment   = form[5][2]


class SportsDiaryApp:
    def __init__(self):
        self.lock = e32.Ao_lock()
        self.exit_flag = False
        appuifw.app.exit_key_handler = self.abort
        self.main_view = appuifw.Listbox([(u"Loading...", u"")],
                                         self.handle_view_entry)
        appuifw.app.body = self.main_view
        self.entry_list = []
        self.menu_add = (u"Add", self.handle_add)
        self.menu_summary = (u"Summary", self.handle_summary)
        self.menu_delete = (u"Delete", self.handle_delete)
        appuifw.app.menu = []

    def initialize_db(self, db_name):
        self.sports_diary = SportsDiary(db_name)

    def run(self):
        while not self.exit_flag:
            self.show_main_view()
            self.lock.wait()
        self.close()

    def close(self):
        appuifw.app.menu = []
        appuifw.app.body = None
        appuifw.app.exit_key_handler = None
        self.sports_diary.close()

    def abort(self):
        self.exit_flag = True
        self.lock.signal()

    def update_entry_list(self):
        self.entry_list = self.sports_diary.get_all_entries()

    def show_main_view(self):
        self.update_entry_list()
        if not self.entry_list:
            content = [(u"(Empty)", u"")]
        else:
            content = [(unicode(time.ctime(item.unixtime())),
                        item.get_sport_text()) for item in
self.entry_list]

        self.main_view.set_list(content)

        if not self.entry_list:
            appuifw.app.menu = [self.menu_add]
        else:
            appuifw.app.menu = [self.menu_add,
                                self.menu_delete,
                                self.menu_summary]

    def handle_add(self):
        new_entry = SportsDiaryEntry()
        data = new_entry.get_form()
        flags = appuifw.FFormEditModeOnly
        f = appuifw.Form(data, flags)
        f.execute()
        new_entry.set_from_form(f)
        self.sports_diary.add(new_entry)
        self.lock.signal()

    def handle_delete(self):
        if self.entry_list:
            index = self.main_view.current()
        if appuifw.query(u"Delete entry?", 'query'):
            self.sports_diary.delete(self.entry_list[index])
        self.lock.signal()
    def handle_summary(self):
        sum = 0
        for e in self.entry_list:
            sum += e.distance
        appuifw.note(u"Total distance is "+str(sum), 'info')

    def handle_view_entry(self):
        if self.entry_list:
            index = self.main_view.current()
            self.show_entry(self.entry_list[index])
        self.lock.signal()

    def show_entry(self, entry):
        data = entry.get_form()
        flags = appuifw.FFormViewModeOnly
        f = appuifw.Form(data, flags)
        f.execute()

def main():
    app = SportsDiaryApp()
    app.initialize_db(u"c:\\SportsDiary.db")
    app.run()

if __name__ == '__main__':
```

```
old_title = appuifw.app.title
try:
    appuifw.app.title = u"Sports diary"
    e32.ao_yield()
    main()
finally:
    appuifw.app.title = old_title
```

Appendix H Source Code for RSS Reader

This program consists of two separate source codes: `rssreader.py` and `simplefeedparser.py`.

H.1 rssreader.py

```
# Copyright (c) 2005 Nokia
# Programming example -- see license agreement for additional rights
# A simple RSS reader application.

import anydbm

import e32
import appuifw
from key_codes import *

class RSSFeed:
    def __init__(self,url,title=None):
        self.url=url
        if title is None:
            title=url
        self.listeners=[]
        self.feed={'title': title,
                   'entries': [],
                   'busy': False}
        self.updating=False
    def listen(self,callback):
        self.listeners.append(callback)
    def _notify_listeners(self,*args):
        for x in self.listeners:
            x(*args)
    def start_update(self):
        if self.feed['busy']:
            appuifw.note(u'Update already in progress','info')
            return
        self.feed['busy']=True
        import thread
        thread.start_new_thread(self._update,())
    def _update(self):
        import dumbfeedparser as feedparser
        newfeed=feedparser.parse(self.url)
        self.feed.update(newfeed)
        self.feed['busy']=False
        self._notify_listeners()
    def __getitem__(self,key):
        return self.feed.__getitem__(key)

class CachingRSSFeed(RSSFeed):
    def __init__(self,cache,url,title=None):
        RSSFeed.__init__(self,url,title)
        self.cache=cache
        if cache.has_key(url):
            self.feed=eval(cache[url])
        self.dirty=False
        RSSFeed.listen(self,self._invalidate_cache)
    def _invalidate_cache(self):
        self.dirty=True
    # This method can't simply be a listener called by the RSSFeed,
    # since that call is done in a different thread and currently the
    # e32dbm module can only be used from the same thread it was
    # opened in.
    def save(self):
        if self.dirty:
            self.cache[self.url]=repr(self.feed)


def format_feed(feed):
    if feed['busy']:
        busyflag='(loading) '
    else:
        busyflag=''
    return unicode('%d: %s%s'%(len(feed['entries']),
                               busyflag,
                               feed['title']))

def handle_screen(mode):
    appuifw.app.screen = mode


class ReaderApp:
    def __init__(self,feedlist):
        self.lock=e32.Ao_lock()
        self.exit_flag=False
        self.old_exit_key_handler=appuifw.app.exit_key_handler
        self.old_app_body=appuifw.app.body
        appuifw.app.exit_key_handler=self.handle_exit
        self.feeds=feedlist
        self.articleviewer=appuifw.Text()
        self.feedmenu=appuifw.Listbox([u''],
                                      self.handle_feedmenu_select)
        self.articlemenu=appuifw.Listbox([u''],
                                         self.handle_articlemenu_select)
        screenmodemenu=(u'Screen mode',
                        ((u'full', lambda:handle_screen('full')),
                         (u'large', lambda:handle_screen('large')),
                         (u'normal', lambda:handle_screen('normal'))))
        self.statemap={
            'feedmenu':
            {'menu':[(u'Update this feed', self.handle_update),
                     (u'Update all feeds', self.handle_update_all),
                     (u'Debug',self.handle_debug),
                     screenmodemenu,
                     (u'Exit',self.abort)],
             'exithandler': self.abort},
            'articlemenu':
            {'menu':[(u'Update this feed', self.handle_update),
                     (u'Update all feeds', self.handle_update_all),
                     screenmodemenu,
                     (u'Exit',self.abort)],
             'exithandler':lambda:self.goto_state('feedmenu')},
            'articleview':
            {'menu':[(u'Next article',self.handle_next),
                     (u'Previous article',self.handle_prev),
                     screenmodemenu,
                     (u'Exit',self.abort)],
             'exithandler':lambda:self.goto_state('articlemenu')}}
        self.articleviewer.bind(EKeyDownArrow,self.handle_downarrow)
        self.articleviewer.bind(EKeyUpArrow,self.handle_uparrow)
        self.articleviewer.bind(EKeyLeftArrow,self.handle_exit)
        self.articlemenu.bind(EKeyRightArrow,
                              self.handle_articlemenu_select)
        self.articlemenu.bind(EKeyLeftArrow,self.handle_exit)
        self.feedmenu.bind(EKeyRightArrow,self.handle_feedmenu_select)
        for k in self.feeds:
            k.listen(self.notify)
        self.goto_state('feedmenu')
    def abort(self):
        self.exit_flag=True
        self.lock.signal()
    def close(self):
        appuifw.app.menu=[]
        appuifw.app.exit_key_handler=self.old_exit_key_handler
        appuifw.app.body=self.old_app_body
    def run(self):
        try:
            while not self.exit_flag:
                self.lock.wait()
                self.refresh()
        finally:
            self.close()
    def notify(self):
        self.lock.signal()
    def refresh(self):
        self.goto_state(self.state)
    def goto_state(self,newstate):
        # Workaround for a Series 60 bug: Prevent the cursor from
        # showing up if the articleviewer widget is not visible.
        self.articleviewer.focus=False
        if newstate=='feedmenu':
            self.feedmenu.set_list(
                [format_feed(x) for x in self.feeds])
            appuifw.app.body=self.feedmenu
            appuifw.app.title=u'RSS reader'
        elif newstate=='articlemenu':
            if len(self.current_feed['entries'])==0:
                if appuifw.query(u'Download articles now?','query'):
                    self.handle_update()
                self.goto_state('feedmenu')
                return
            self.articlemenu.set_list(
                [self.format_article_title(x)
                 for x in self.current_feed['entries']])
            appuifw.app.body=self.articlemenu
            appuifw.app.title=format_feed(self.current_feed)
        elif newstate=='articleview':
            self.articleviewer.clear()
            self.articleviewer.add(
                self.format_title_in_article(self.current_article()))
            self.articleviewer.add(
                self.format_article(self.current_article()))
            self.articleviewer.set_pos(0)
            appuifw.app.body=self.articleviewer
            appuifw.app.title=self.format_article_title(
                self.current_article())
        else:
            raise RuntimeError("Invalid state %s"%state)
        appuifw.app.menu=self.statemap[newstate]['menu']
        self.state=newstate
    def current_article(self):
        return self.current_feed['entries'][self.current_article_index]
    def handle_update(self):
        if self.state=='feedmenu':
            self.current_feed=self.feeds[self.feedmenu.current()]
        self.current_feed.start_update()
        self.refresh()
    def handle_update_all(self):
        for k in self.feeds:
            if not k['busy']:
                k.start_update()
        self.refresh()
    def handle_feedmenu_select(self):
        self.current_feed=self.feeds[self.feedmenu.current()]
        self.goto_state('articlemenu')
    def handle_articlemenu_select(self):
        self.current_article_index=self.articlemenu.current()
        self.goto_state('articleview')
    def handle_debug(self):
        import btconsole
        btconsole.run('Entering debug mode.',locals())
    def handle_next(self):
        if (self.current_article_index >=
            len(self.current_feed['entries'])-1):
            return
        self.current_article_index += 1
        self.refresh()
    def handle_prev(self):
        if self.current_article_index == 0:
            return
        self.current_article_index -= 1
        self.refresh()
    def handle_downarrow(self):
        article_length=self.articleviewer.len()
        cursor_pos=self.articleviewer.get_pos()
        if cursor_pos==article_length:
            self.handle_next()
        else:
            self.articleviewer.set_pos(min(cursor_pos+100,
                                           article_length))
    def handle_uparrow(self):
        cursor_pos=self.articleviewer.get_pos()
        if cursor_pos==0:
            self.handle_prev()
            self.articleviewer.set_pos(self.articleviewer.len())
        else:
            self.articleviewer.set_pos(max(cursor_pos-100,0))
    def format_title_in_article(self, article):
        self.articleviewer.highlight_color = (255,240,80)
        self.articleviewer.style = (appuifw.STYLE_UNDERLINE|
                                    appuifw.HIGHLIGHT_ROUNDED)
        self.articleviewer.font = 'title'
        self.articleviewer.color = (0,0,255)
        return unicode("%(title)s\n"%article)

    def format_article(self, article):
        self.articleviewer.highlight_color = (0,0,0)
        self.articleviewer.style = 0
        self.articleviewer.font = 'normal'
        self.articleviewer.color = (0,0,0)
        return unicode("%(summary)s"%article)

    def format_article_title(self, article):
        return unicode("%(title)s"%article)
    def handle_exit(self):
        self.statemap[self.state]['exithandler']()

class DummyFeed:
    def __init__(self,data): self.data=data
    def listen(self,callback): pass
    def start_update(self): pass
    def __getitem__(self,key): return self.data.__getitem__(key)
    def save(self): pass
dummyfeed=DummyFeed({'title': 'Dummy feed',
                     'entries': [{'title':'Dummy story',
                                  'summary':'Blah blah blah.'},
                                 {'title':'Another dummy story',
                                  'summary':'Foo, bar and baz.'}],
                     'busy': False})

def main():
    old_title=appuifw.app.title
    appuifw.app.title=u'RSS reader'
    cache=anydbm.open(u'c:\\rsscache','c')
    feeds=[ CachingRSSFeed(url='http://slashdot.org/index.rss',
                           title='Slashdot',
                           cache=cache),

CachingRSSFeed(url='http://news.bbc.co.uk/rss/newsonline_world_edition/f
ront_page/rss091.xml',
                           title='BBC',
                           cache=cache),
            dummyfeed]
    app = ReaderApp(feeds)
    # Import heavyweight modules in the background to improve
application
    # startup time.
    def import_modules():
        import dumbfeedparser as feedparser
        import btconsole
    import thread
    thread.start_new_thread(import_modules,())
    try:
        app.run()
    finally:
        for feed in feeds:
            feed.save()
        cache.close()
        appuifw.app.title=old_title

if __name__=='__main__':
    main()
```

H.2 simplefeedparser.py

```
# Copyright (c) 2004 Nokia
# Programming example -- see license agreement for additional rights
# A simple and limited RSS feed parser used in the RSS reader example.

import re
import urllib

def parse(url):
    return parse_feed(urllib.urlopen(url).read())

def parse_feed(text):
    feed={}
    items=[]
    currentitem=[{}]

    def clean_entities(text): return re.sub('&[#0-9a-z]+;','?',text)
    def clean_lf(text): return re.sub('[\n\t\r]',' ',text)

    def end_a(tag,content): write('LINK(%s)'%gettext())
    def start_item(tag,content):
        gettext()
        write(content)
        currentitem[0]={}
    def end_item(tag,content):
        items.append(currentitem[0])
        currentitem[0]={}
    def end_link(tag,content):
        if within('item'):
            currentitem[0]['link']=gettext()
    def end_description(tag,content):
        if within('item'):
            currentitem[0]['summary']=clean_entities(gettext())
    def end_title(tag,content):
        text=clean_lf(gettext()).strip()
        if within('item'):
            currentitem[0]['title']=text
        elif parentis('channel'):
            feed['title']=text

    tagre=re.compile('([^ \n\t]+)(.*)>(.*)',re.S)
    tagpath=[]
    textbuffer=[[]]
    assumed_encoding='latin-1'
    lines=text.split('<')
    def start_default(tag,content): write(content)
    def end_default(tag,content): pass
    def tag_default(tag,content): pass
    def write(text): textbuffer[0].append(text)
    def gettext():
        text=''.join(textbuffer[0])
        textbuffer[0]=[]
        return unicode(text,assumed_encoding)
    def current_tag(): return tagpath[-1]
    def current_path(): return '/'.join(tagpath)
    def within(tag): return tag in tagpath
    def parentis(tag): return current_tag()==tag
    for k in lines:
        m=tagre.match(k)
        if m:
            (tag,attributes,content)=m.groups()
            if tag.startswith('?'):
                continue
            if tag.startswith('/'):
                tagname=tag[1:]
                handler='end_%s'%tagname
                generic_handler=end_default
                if current_tag() != tagname:
                    pass # Unbalanced tags, just ignore for now.
                del tagpath[-1]
            elif tag.endswith('/'):
                tagname=tag[0:-1]
                handler='tag_%s'%tagname
                generic_handler=tag_default
            else:
                tagname=tag
                handler='start_%s'%tagname
                generic_handler=start_default
                tagpath.append(tagname)
            locals().get(handler,generic_handler)(tagname,content)
        else:
            pass # Malformed line, just ignore.

    feed['entries']=items
    return feed
```

Appendix I Source Code for default.py

```
#
# default.py
#
# The default script run by the "Python" application in Series 60 Python
# environment. Offers menu options for running scripts that are found in
# application's directory, or in the \my -directory below it (this is
# where the application manager copies the plain Python scripts sent to
# device's inbox), as well as for launching interactive Python console.
#
# Copyright (c) 2004 Nokia. All rights reserved.
#

import sys
import os
import appuifw
import series60_console

def query_and_exec():
    def is_py(x):
        return os.path.splitext(x)[1] == '.py'

    my_script_dir = os.path.join(this_dir,'my')
    script_list = []

    if os.path.exists(my_script_dir):
        script_list = map(lambda x: os.path.join('my',x),\
                              filter(is_py, os.listdir(my_script_dir)))

    script_list += filter(is_py, os.listdir(this_dir))
    index = appuifw.selection_list(map(unicode, script_list))
    if index >= 0:
        execfile(os.path.join(this_dir, script_list[index]), globals())

def exec_interactive():
    import interactive_console
    interactive_console.Py_console(my_console).interactive_loop()

def exec_btconsole():
    import btconsole
    btconsole.main()

def menu_action(f):
    appuifw.app.menu = []
    saved_exit_key_handler = appuifw.app.exit_key_handler
    try:
        try:
            f()
        finally:
            appuifw.app.exit_key_handler = saved_exit_key_handler
            appuifw.app.title = u'Python'
            init_options_menu()
            appuifw.app.body = my_console.text
            appuifw.app.screen='normal'
            sys.stderr = sys.stdout = my_console
    except:
        import traceback
        traceback.print_exc()

def init_options_menu():
    appuifw.app.menu = [(u"Run script",\
                         lambda: menu_action(query_and_exec)),
                        (u"Interactive console",\
                         lambda: menu_action(exec_interactive)),\
                        (u"Bluetooth console",\
                         lambda: menu_action(exec_btconsole)),\
                        (u"About Python",\
                         lambda: appuifw.note(u"See www.python.org for
more information.", "info"))]

this_dir = os.path.split(appuifw.app.full_name())[0]
my_console = series60_console.Console()
appuifw.app.body = my_console.text
sys.stderr = sys.stdout = my_console
#from e32 import _stdo
#_stdo(u'c:\\python_error.log')         # low-level error output
init_options_menu()
print copyright
```

Appendix J Source Code for Example Extension

J.1 elemlist.cpp

```
/*
   Copyright (c) 2005 Nokia
   Programming example -- see license agreement for additional rights
   A simple extension used in the Porting an Extension example.
*/
/*
   This extension module implements a new native type, the "cons
   cell", that is very similar to the cons cells used in Lisp.

   This code illustrates some of the issues that arise when creating
   extensions for Python for Series 60. The code is derived from the
   example extension written by Alex Martelli for the Python
   Cookbook. The original code is licensed under the Python license,
   which is available at http://www.python.org/license.

   All parts that had to be modified from the original have
   been clearly marked. A summary of modifications:

   - Since Symbian DLLs do not (properly) support global writable
   data, the type object is allocated dynamically and filled in from a
   const template. Also, the function table for the module has been
   declared const.

   - The macro versions of memory allocation routines (PyObject_NEW,
   PyObject_DEL and others) are not supported in Python for Series 60
   1.0, so the non-macro versions, PyObject_New, PyObject_Del must be
   used instead.

   - The file has been compiled with the C++ compiler, to make it
   possible to include Symbian headers.
*/

#include "Python.h"

/*******************************************************
 This include file declares the SPyGetGlobalString and
 SPySetGlobalString functions: */
#include "symbian_python_ext_util.h"
/* Standard Symbian definitions: */
#include <e32std.h>
/*******************************************************/

/* type-definition & utility-macros */
typedef struct {
    PyObject_HEAD
    PyObject *car, *cdr;
} cons_cell;


/*******************************************************
   Original definition:
   staticforward PyTypeObject cons_type;

   Symbian does not support writable global data in DLLs, so
   this type object has to be stored in another way. We
   choose to allocate it dynamically in the module init
   function and to bind it to a global name, so that we can
   access it with the SPyGetGlobalString function. For
   convenience, we define a macro that encapsulates the use
   of that function. Naming a macro in all lowercase
   violates the standard naming convention for macros, but
   allows you to keep the code that handles the type
   unchanged, which may be convenient if the same source
   code is used on multiple platforms. You will have to
   decide for yourself if this is too unsavory for your
   tastes. */
#define cons_type (*(PyTypeObject *)SPyGetGlobalString("consType"))
/*********************************************************/


/* a typetesting macro (we do not use it here) */
#define is_cons(v)  ((v)->ob_type == &cons_type)
/* macros to access car & cdr, both as lvalues & rvalues */
#define carof(v)  (((cons_cell*)(v))->car)
#define cdrof(v)  (((cons_cell*)(v))->cdr)

/* ctor (factory-function) and dtor */
static cons_cell*
cons_new(PyObject *car, PyObject *cdr)
{
    /*********************************************************
     Original code:
       cons_cell *cons = PyObject_NEW(cons_cell, &cons_type);
     The macro versions of memory allocation routines
     (PyObject_NEW, PyObject_DEL and others) are not supported
     in Python for Series 60 1.0, so the non-macro versions,
     PyObject_New, PyObject_Del must be used instead.

     The Python documentation states that the use of these macros in
     extensions is bad practice in any case, since it ties the
     extension to the behaviour of the interpreter in unpredictable
     ways. */
    cons_cell *cons = PyObject_New(cons_cell, &cons_type);
    /*********************************************************/
    if(cons) {
        cons->car = car; Py_INCREF(car);  /* INCREF when holding a
PyObject* */
        cons->cdr = cdr; Py_INCREF(cdr);  /* ditto */
    }
    return cons;
}
static void
cons_dealloc(cons_cell* cons)
{
    /* DECREF when releasing previously-held PyObject*'s */
    Py_DECREF(carof(cons)); Py_DECREF(cdrof(cons));
    /*********************************************************
     Original code:
       PyObject_DEL(cons);
     See the note on PyObject_NEW.*/
    PyObject_Del(cons);
    /*********************************************************/
}

/* Python type-object */

/*********************************************************
   Original definition:
   statichere PyTypeObject cons_type = {

   As mentioned above, Symbian does not support _writable_
   global data in DLLs, so we store this partially
   initialized type object as constant data. In the module
   init function this is copied to a dynamically allocated,
   writable memory region. Note the name change to avoid
   clashing with the macro cons_type defined above. */
static const PyTypeObject cons_type_template = {
/*********************************************************/
    PyObject_HEAD_INIT(0)    /* initialize to 0 to ensure Win32
portability */
    0,                 /*ob_size*/
    "cons",            /*tp_name*/
    sizeof(cons_cell), /*tp_basicsize*/
    0,                 /*tp_itemsize*/
    /* methods */
    (destructor)cons_dealloc, /*tp_dealloc*/
    /* implied by ISO C: all zeros thereafter */
};

/* module-functions */
static PyObject*
cons(PyObject *self, PyObject *args)    /* the exposed factory-function
*/
{
    PyObject *car, *cdr;
    if(!PyArg_ParseTuple(args, "OO", &car, &cdr))
        return 0;
    return (PyObject*)cons_new(car, cdr);
}
static PyObject*
car(PyObject *self, PyObject *args)     /* car-accessor */
{
    PyObject *cons;
    if(!PyArg_ParseTuple(args, "O!", &cons_type, &cons))  /* type-
checked */
        return 0;
    return Py_BuildValue("O", carof(cons));
}
static PyObject*
cdr(PyObject *self, PyObject *args)     /* cdr-accessor */
{
    PyObject *cons;
    if(!PyArg_ParseTuple(args, "O!", &cons_type, &cons))  /* type-
checked */
        return 0;
    return Py_BuildValue("O", cdrof(cons));
}
/*******************************************************
   Original definition:
   static PyMethodDef elemlist_methods[] = {

   Since no-one needs to write to this particular array,
   we can make the code work simply by adding "const".
   There is no need to copy the data. */
static const PyMethodDef elemlist_methods[] = {
/*******************************************************/
    {"cons",   cons,   METH_VARARGS},
    {"car",    car,    METH_VARARGS},
    {"cdr",    cdr,    METH_VARARGS},
    {0, 0}
};

/* module entry-point (module-initialization) function */
/*******************************************************
   Original definition:
   void initelemlist(void)

   We have to explicitly state that this function must be
   included in the public function list of the DLL. Symbian
   DLLs do not include names of the exported functions. The
   program that loads a DLL has to know the the index of the
   function in the function table, commonly known as the
   ordinal of the function, to call the functions in the
   DLL.

   The initialization function for a Python module _must_ be
   exported at ordinal 1. If the module exports just the
   initializer function, then there is nothing to worry
   about. If you also export a module finalizer function,
   you will have to make sure that the initializer is
   exported at ordinal 1 and the finalizer at ordinal 2.

   This module has just the initializer function.

   Also, the function has to be declared extern "C":*/
extern "C" {
DL_EXPORT(void) initelemlist(void)
/*******************************************************/
{
    /* Create the module and add the functions */
    /*******************************************************
      Original code:
        PyObject *m = Py_InitModule("elemlist", elemlist_methods);
      The Python/C API is unfortunately not quite const-correct, so
      we need to add a cast here to make the compiler happy:*/
    PyObject *m = Py_InitModule("elemlist",
(PyMethodDef*)elemlist_methods);
    /*******************************************************/

    /* Finish initializing the type-objects */

    /*******************************************************
     Allocate storage for the type object, fill it in
     from the constant template and bind it to a name in the
     module namespace: */
    PyTypeObject *consTypeObject=PyObject_New(PyTypeObject,
&PyType_Type);
    *consTypeObject=cons_type_template;
    SPyAddGlobalString("consType", (PyObject *)consTypeObject);
    /*******************************************************/

    cons_type.ob_type = &PyType_Type;
}
}

/*******************************************************
 This function is mandatory in Symbian DLL's. */

GLDEF_C TInt E32Dll(TDllReason)
{
  return KErrNone;
}
/*******************************************************/
```

J.2 bld.inf

```
PRJ_PLATFORMS
wins winscw armi

PRJ_MMPFILES
elemlist.mmp
```

J.3 elemlist.mmp

```
TARGETTYPE  dll
TARGET      elemlist.pyd
TARGETPATH  \system\libs

NOSTRICTDEF
DEFFILE     elemlist.frz

SYSTEMINCLUDE  \epoc32\include
SYSTEMINCLUDE  \epoc32\include\libc

USERINCLUDE    \python-port-s60\symbian_python\Symbian
USERINCLUDE    \python-port-s60\symbian_python\Include
USERINCLUDE    \python-port-s60\symbian_python\Python

LIBRARY  python222.lib
LIBRARY  euser.lib
LIBRARY  estlib.lib /* Necessary only if you use the C standard library
*/

SOURCE      elemlist.cpp
```

Appendix K Contacts and Calendar Examples

K.1 Print Entries and Their Total Number in the Default Contacts Database

```
db = contacts.open()
for entry_id in db:
  print db[entry_id]
print u'number of entries:%i'%len(db)
```

K.2 Modifying a Contact

```
# open the database..
db = contacts.open()

# add new contact
contact = db.add_contact()
contact.add_field('first_name', value=u'John')
contact.add_field('last_name', u'Doe')
contact.add_field('mobile_number','76476548','work')
contact.commit()

print u'the contact at first:%s'%contact

# modify the contact
contact.find('first_name')[0].value='Henry'

print u'the contact now:%s'%contact

# delete the first name field
del contact[contact.find('first_name')[0].index]

print u'and now:%s'%contact

# delete the contact
del db[contact.id]
```

K.3 Using Calendar Entry's Properties, etc.

```
db = calendar.open()

week = 7*24*60*60
hour = 60*60
minute = 60
now = time.time()

print u'entries in db:%i'%len(db)
print u'add an appointment..'
new_entry = db.add_appointment() # new appointment (note that at first
the autocommit is off and changes must be explisitly committed).
new_entry.set_time(now+week,now+week+hour)
new_entry.alarm=now+week-5*minute
new_entry.content=u'the meeting'
new_entry.location=u'conference room 01'
new_entry.replication="private"
new_entry.priority=1 # note that in sdk 1.2 only Todo's have priority.
new_entry.commit() # commit because autocommit is off.
print u'entries in db now:%i'%len(db)
print u'**entry\'s data**'
print u'id:%i'%new_entry.id
print u'content:%s'%new_entry.content
print u'location:%s'%new_entry.location
print u'start_time:%s'%time.ctime(new_entry.start_time)
print u'end_time:%s'%time.ctime(new_entry.end_time)
print u'last modified:%s'%time.ctime(new_entry.last_modified)
print u'alarm datetime:%s'%time.ctime(new_entry.alarm)
print u'replication:%s'%new_entry.replication
print u'priority:%s'%new_entry.priority
print u'crossed out:%s'%new_entry.crossed_out
print u'--------'

# to cross out the entry
new_entry.crossed_out=1 # note that autocommit is now on.
print "after crossing out:"
print u'crossed out:%s'%new_entry.crossed_out
print u'alarm:%s'%str(new_entry.alarm)
print ""

print u'delete the entry..'
del db[new_entry.id]
print u'entries in db now:%i'%len(db)
print ""

# add todo entry
print u'add a todo..'
new_entry = db.add_todo() # new todo
new_entry.set_time(now+week)
new_entry.content=u'the things todo'
new_entry.location=u'work'
new_entry.replication="private"
new_entry.priority=2
new_entry.commit()

print u'entries in db now:%i'%len(db)
print u'**entry\'s data**'
print u'id:%i'%new_entry.id
print u'content:%s'%new_entry.content
print u'location:%s'%new_entry.location
print u'start_time:%s'%time.ctime(new_entry.start_time)
print u'end_time:%s'%time.ctime(new_entry.end_time)
print u'last modified:%s'%time.ctime(new_entry.last_modified)
print u'replication:%s'%new_entry.replication
print u'priority:%s'%new_entry.priority
print u'crossed out:%s'%new_entry.crossed_out
print u'--------'

# to cross out the entry
new_entry.cross_out_time=time.time()
print "after crossing out:"
print u'crossed out:%s'%new_entry.crossed_out
print u'cross out time:%s'%time.ctime(new_entry.cross_out_time)
print ""

# to make the todo entry undated.
print "after making undated:"
new_entry.set_time(None)
print u'start_time:%s'%new_entry.start_time
print u'end_time:%s'%new_entry.end_time

print u'to delete the entry..'
del db[new_entry.id]
print u'entries in db now:%i'%len(db)
```

K.4 Todo Lists

```
week = 7*24*60*60
db = calendar.open()

td=db.add_todo()
td.content=u'a test todo'
td.set_time(time.time(),time.time())
td.commit()

print "todo lists:"
for list_id in db.todo_lists:
  print "list id: %d"%list_id
  print "list name: %s"%db.todo_lists[list_id].name
  for entry_id in db.todo_lists[list_id]:
    print "todo (id) in the list: %d"%entry_id

print "default todo list: %d"%db.todo_lists.default_list

# create new todo list
list_id = db.add_todo_list(u'new todo list')

print "new todo list name: %s"%db.todo_lists[list_id].name

# rename it
db.todo_lists[list_id].name=u'renamed new todo list'

print "todo list name after renaming: %s"%db.todo_lists[list_id].name

# remove the created todo list (note that all todo’s in the list are
also removed from the database)

del db.todo_lists[list_id]
```

Appendix L Source Code for Ball

```
#
# ball.py
# Copyright (c) 2005 Nokia. All rights reserved.
# Programming example -- see license agreement for additional rights
# A simple application used in the graphics support example.

import appuifw
from graphics import *
import e32
from key_codes import *

class Keyboard(object):
    def __init__(self,onevent=lambda:None):
        self._keyboard_state={}
        self._downs={}
        self._onevent=onevent
    def handle_event(self,event):
        if event['type'] == appuifw.EEventKeyDown:
            code=event['scancode']
            if not self.is_down(code):
                self._downs[code]=self._downs.get(code,0)+1
            self._keyboard_state[code]=1
        elif event['type'] == appuifw.EEventKeyUp:
            self._keyboard_state[event['scancode']]=0
        self._onevent()
    def is_down(self,scancode):
        return self._keyboard_state.get(scancode,0)
    def pressed(self,scancode):
        if self._downs.get(scancode,0):
            self._downs[scancode]-=1
            return True
        return False
keyboard=Keyboard()

appuifw.app.screen='full'
img=None
def handle_redraw(rect):
    if img:
        canvas.blit(img)
appuifw.app.body=canvas=appuifw.Canvas(
    event_callback=keyboard.handle_event,
    redraw_callback=handle_redraw)
img=Image.new(canvas.size)

running=1
def quit():
    global running
    running=0
appuifw.app.exit_key_handler=quit

location=[img.size[0]/2,img.size[1]/2]
speed=[0.,0.]
blobsize=16
xs,ys=img.size[0]-blobsize,img.size[1]-blobsize
gravity=0.03
acceleration=0.05

import time
start_time=time.clock()
n_frames=0
while running:
    img.clear(0)
    img.text((0,14),u'Use arrows to move ball',0xffffff)
    img.point((location[0]+blobsize/2,location[1]+blobsize/2),
              0x00ff00,width=blobsize)
    handle_redraw(())
```

```
        e32.ao_yield()
        speed[0]*=0.999
        speed[1]*=0.999
        speed[1]+=gravity
        location[0]+=speed[0]
        location[1]+=speed[1]
        if location[0]>xs:
            location[0]=xs-(location[0]-xs)
            speed[0]=-0.80*speed[0]
            speed[1]=0.90*speed[1]
        if location[0]<0:
            location[0]=-location[0]
            speed[0]=-0.80*speed[0]
            speed[1]=0.90*speed[1]
        if location[1]>ys:
            location[1]=ys-(location[1]-ys)
            speed[0]=0.90*speed[0]
            speed[1]=-0.80*speed[1]
        if location[1]<0:
            location[1]=-location[1]
            speed[0]=0.90*speed[0]
            speed[1]=-0.80*speed[1]

        if keyboard.is_down(EScancodeLeftArrow):  speed[0] -= acceleration
        if keyboard.is_down(EScancodeRightArrow): speed[0] += acceleration
        if keyboard.is_down(EScancodeDownArrow):  speed[1] += acceleration
        if keyboard.is_down(EScancodeUpArrow):    speed[1] -= acceleration
        if keyboard.pressed(EScancodeHash):
            filename=u'e:\\screenshot.png'
            canvas.text((0,32),u'Saving screenshot to:',fill=0xffff00)
            canvas.text((0,48),filename,fill=0xffff00)
            img.save(filename)

        n_frames+=1
    end_time=time.clock()
    total=end_time-start_time

    print "%d frames, %f seconds, %f FPS, %f ms/frame."%(n_frames,total,
                                                         n_frames/total,
                                                       total/n_frames*1000.)
```