FORUM NOKIA

Python for Series 60 Platform API Reference

Version 1.2; September 28, 2005

Python for Series 60 Platform

NOKIA

Disclaimer

The information in this document is provided "as is," with no warranties whatsoever, including any warranty of merchantability, fitness for any particular purpose, or any warranty otherwise arising out of any proposal, specification, or sample. Furthermore, information provided in this document is preliminary, and may be changed substantially prior to final release. This document is provided for informational purposes only.

Nokia Corporation disclaims all liability, including liability for infringement of any proprietary rights, relating to implementation of information presented in this document. Nokia Corporation does not warrant or represent that such use will not infringe such rights.

Nokia Corporation retains the right to make changes to this specification at any time, without notice.

License

A license is hereby granted to download and print a copy of this specification for personal use only. No other license to any other intellectual property rights is granted herein.

Contents

Change History

December 10, 2004	Version 1.0	Initial document release.
June 29, 2005	Version 1.1.5	Sections 1.3, 3.3, 4.5, 4.6, 4.7, 5.6, 5.8, 6, 12, 13, 14, 15, 16, and 17 added. Sections 1, 3.1, 3.2, 5.1, 5.2, 5.4, 5.5, 7.1, and 18.2 updated.
September 28, 2005	Version 1.2	Section 4.7 added. Sections 1.3, 5.1, 5.2, 5.4, 5.5, 5.6, 13, and 19 updated.

1 Introduction

The Python for Series 60 Platform (Python for Series 60) 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 [5]* contains information on how to install Python for Series 60 and how to write your first program.
- This document contains API and other reference material.
- *Programming with Python for Series 60 Platform [6]* contains code examples and programming patterns for Series 60 devices that can be used as a basis for programs.

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 documentation [4]*)
- 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 the 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 *[9]* 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 the information required by developers to create applications that use Python for Series 60, and some advice on extending the platform.

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 (http://www.python.org) and the basics of using Python for Series 60 (see *Getting Started with Python for Series 60 Platform [5]*).

1.3 New in Release 1.2

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

- Section 4.7, *Scalable User Interface* has been added.
- There are some general updates in 5.1, *Module Level Functions* and 5.2, *Application Type*.
- Section 5.4, *Text Type* has been updated to include a `delete` method.
- Sections 5.5, *Listbox Type* and 5.6, Icon Type have been updated with SVG-T support information.
- Chapter 13, *camera Module* has been updated with new functions.

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.

1.5 Naming Conventions

Most names of the type `ESomething` typically indicate a constant defined by the Symbian SDK. More information about these constants can be found in the Symbian SDK documentation.

2 API Summary

All built-in object types of the Python language are supported in the Series 60 environment. The rest of the programming interfaces are implemented by various library modules as summarized in this chapter.

2.1 Python Standard Library

Python for Series 60 platform distribution does not include all of the Python's standard and optional library modules to save storage space in the phone. Nevertheless, many of the excluded modules also work in the Series 60 Python environment without any modifications. Some modules are included in the SDK version but not installed in the phone. For a summary of supported library modules, see *Appendix A*, *Python Library Module Support*.

When Python, available at http://www.python.org, is installed on a PC, the library modules are by default located in `\Python22\Lib` on Windows and in `/usr/lib/python2.2` on Linux. The Python library modules' APIs are documented in *G. van Rossum, and F.L. Drake, Jr., editor. [Python] Library Reference. [1]*

Python for Series 60 extends some standard modules. These extensions are described in this document, see Chapter *18*, *Extensions to Standard Library Modules*.

2.2 Python for Series 60 Extensions

There are two kinds of native C++ extensions in the Python for Series 60 Platform: built-in extensions and dynamically loadable extensions.

2.2.1 Built-in extensions

There are two built-in extensions in the Python for Series 60 package:

- The `e32` extension module is built into the Python interpreter on Symbian OS, and implements interfaces to special Symbian OS Platform services that are not accessible via Python standard library modules.
- The `appuifw` module for Python for Series 60 Platform offers UI application framework related Python interfaces.

2.2.2 Dynamically loadable extensions

These dynamically loadable extension modules provide proprietary APIs to Series 60 Platform's services: `graphics` (see Chapter *6*, *graphics Module*), `e32db` (see Chapter *8*, *e32db Module*), `messaging` (see Chapter *10*, *messaging Module*), `location` (see Chapter *11*, *location Module*), `sysinfo` (see Chapter *12*, *sysinfo Module*), `camera` (see Chapter *13*, *camera Module*), `audio` (see Chapter *14*, *audio Module*), `telephone` (see Chapter *15*, *telephone Module*), `calendar` (see Chapter *16*, *calendar Module*), and `contacts` (see Chapter *17*, *contacts Module*).

2.3 Third-Party Extensions

It is also possible to write your own Python extensions. Series 60 related extensions to Python/C API are described in *Appendix B*, *Extensions to C API*. For some further guidelines on writing extensions in C/C++, see *Appendix C*, *Extending Series 60 Python*. In addition, for an example on porting a simple extension to Series 60, see *Programming with Python for Series 60 Platform* [*6*].

3 Python for Series 60 UI Programming Model

This chapter gives an outline of the conceptual model of UI application programming that underlies the APIs described in the following chapters.

3.1 Overview

Figure 1 shows the Python for Series 60 environment for UI application programming. The built-in `appuifw` Python module provides an interface to the Series 60 framework.

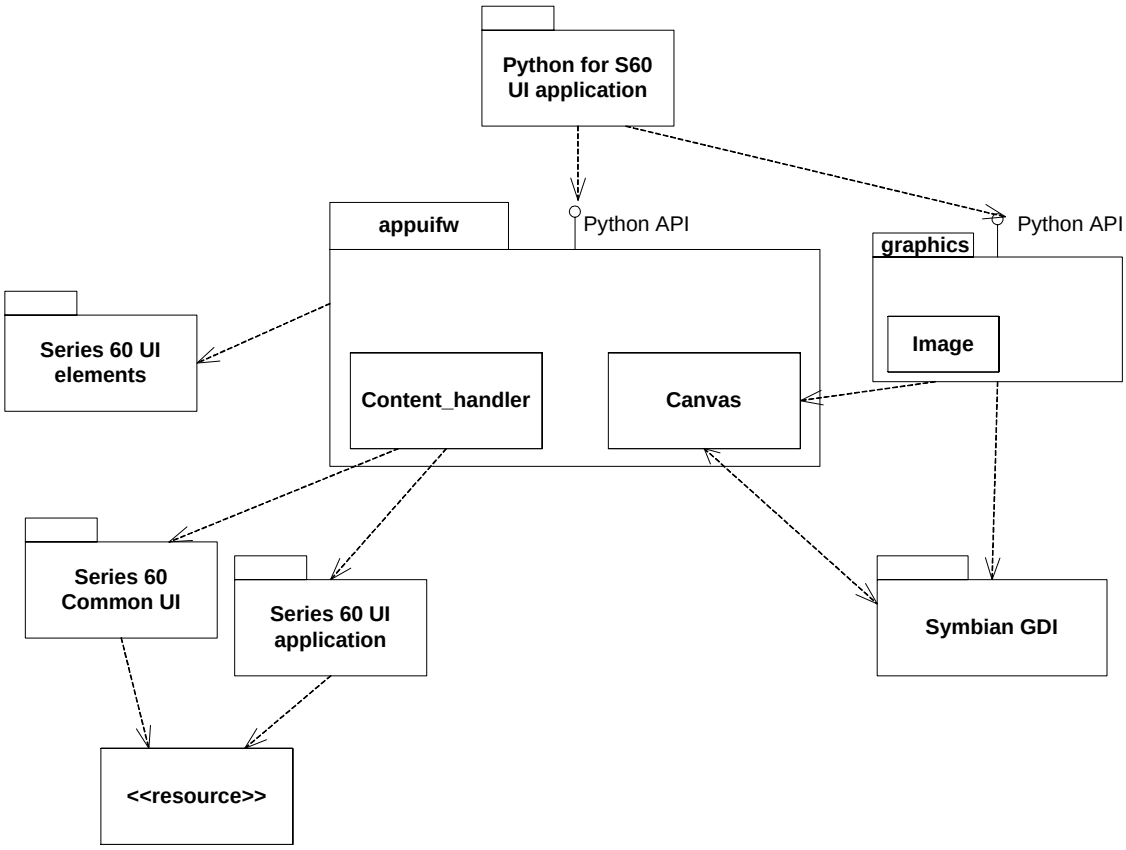

Figure 1: UI application programming

The `appuifw.Content_handler` object type facilitates interfacing to other UI applications and common high-level UI components. It is based on the notion that designated handlers can reduce UI application interaction to operations on MIME-type content.

3.2 Basics of appuifw Module

Figure 2 shows the layout of a Series 60 application UI in the **normal** screen mode and a summary of how it relates to the services available at the `appuifw` API. For alternative layouts, see Figure 4.

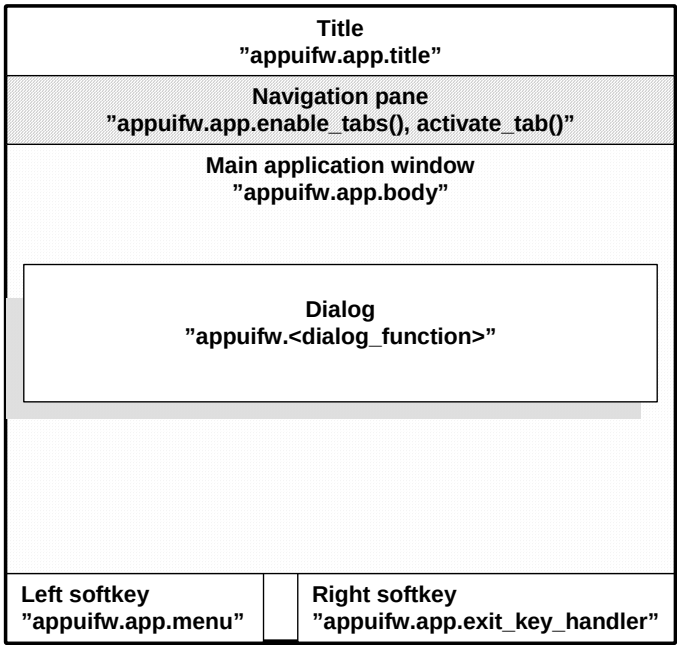

Figure 2: UI layout when `appuifw.app.screen` is set to `'normal'`

The main application window may be set up to be occupied by a UI control.

A multi-view application can show the different views as tabs in the navigation pane and react as the users navigate between tabs.

Dialogs always take precedence over the usual UI controls and appear on top of them.

3.3 appuifw UI Controls

The UI controls are implemented as Python types. These types are available:

- `Text`
- `Listbox`
- `Canvas`

UI controls appear on the screen as soon as an instance of the corresponding Python type is created and set to the body field (`app.body`) of the current application UI.

`Form` is a versatile dialog implemented as a type.

The following dialogs are implemented as functions:

- `note`
- `query`
- `multi_query`
- `selection_list`

- `multi_selection_list`
- `popup_menu`

A dialog becomes visible as soon as the corresponding Python function has been called. The function returns with the eventual user input or information on the cancellation of the dialog. `Form` is an exception; it is shown when its `execute` method is called.

3.4 Softkeys

The softkeys are managed by the underlying Series 60 Platform. When no dialog is visible, the right softkey is bound to application exit and the left one represents an **Options** menu. Python for Series 60 offers an interface for manipulating the menu (see Section *5.2*, *Application Type*, `menu`) and for binding the **Exit** key to a Python-callable object.

The native code that implements a dialog also manages the softkeys of the dialog, typically **OK** and **Cancel**. When the user input needs to be validated before accepting it and dismissing the dialog, it is best to use `Form`.

4 Selected Issues on Python Programming for Series 60

The following issues must be considered when using Python on Series 60.

4.1 Concurrency Aspects

The thread that initializes the Python interpreter becomes the main Python thread. This is usually the main thread of a UI application. When an application written in Python launches, the Symbian platform infrastructure creates the main UI thread that starts the Python environment. If a Python program is started as a server with `e32.start_server`, then the Python main thread is not a UI thread.

It is possible to launch new threads via the services of `thread` module. Examples of such situations could be to overcome eventual problems with the fixed, relatively small stack size of the main UI application thread; or to perform some background processing while still keeping the UI responsive. These new threads are not allowed to directly manipulate the UI; in other words, they may not use the `appuifw` module.

Because of the limitations of the Python interpreter's final cleanup, Python applications on the Symbian OS should be designed in such a way that the main thread is the last thread alive.

A facility called *active object* is used extensively on the Symbian OS to implement co-operative, non-preemptive scheduling within operating system threads. This facility is also utilized with native APIs. A Python programmer is exposed to related concurrency issues particularly in UI programming. Preserving the responsiveness of the UI with the help of active objects needs to be considered when designing the application logic. At the same time it is necessary to take into account the resulting concurrent behavior within the application when active objects are used. While the main execution path of a UI script is blocked in wait for an active object to complete – either explicitly as a result of using `e32.Ao_lock`, or indirectly within some other Python API implementation – the UI-related callbacks may still get called.

The standard `thread.lock` cannot normally be used for synchronization in the UI application main thread, as it blocks the UI event handling that takes place in the same thread context. The Symbian active object based synchronization service called `e32.Ao_lock` has been implemented to overcome this problem. The main thread can wait in this lock, while the UI remains responsive.

Python for Series 60 tries to minimize the unwanted exposure of a Python programmer to the active objects of the Symbian OS. The programmer may choose to implement the eventual concurrent behavior of the application with normal threads. However, certain active object based facilities are offered as an option in the `e32` module.

4.2 Current Series 60 Python Script Execution Environment

The current options for installing Python scripts to a Series 60 device are: a stand-alone installation to the device's main application menu, and an installation to a folder hierarchy maintained by the Python execution environment. For more details on this topic, see *Programming with Python for Series 60 Platform [6]*. In the first case the script application is launched via application menu, and it executes in its own process context. The latter case is suitable for development, testing, and trying out new scripts.

The Python execution environment delivered with Python for Series 60 package has itself been written in Python. It is a collection of scripts that offer an interactive Python console and a possibility to execute scripts located in the directory of the execution environment. Due to this kind of design the scripts are not fully isolated from each other. This means that any changes a script makes in the

shared execution environment are visible to other scripts as well. This may be helpful during the development of a script suite, as long as care is taken to avoid unwanted interference between scripts.

For some special issues to consider when writing Python scripts to be run from the current Python execution environment, see *Programming with Python for Series 60 Platform [6]*. These include the arrangements for standard output and the maintenance of the **Options** menu contents.

4.3 Standard I/O Streams

The standard Python I/O streams in the `sys` module are by default connected to underlying C STDLIB's `stdio` streams that in turn are terminated by dummy file descriptors. Usually Python scripts set the I/O streams suitably by manipulating them at Python level via `sys` module interface. The `e32` extension module offers a Python interface for attaching to C STDLIB's output streams, but this service is only recommended for debugging purposes. The `e32._stdo` function takes as its argument the name of the file where C STDLIB's `stdout` and `stderr` are to be redirected. This makes it possible to capture the low-level error output when the Python interpreter has detected a fatal error and aborts.

4.4 Usage of Unicode

No changes have been made to the standard library modules with regard to string argument and return value types. Series 60 extensions generally accept both plain strings and Unicode strings as arguments, but they return only Unicode strings. APIs that take string arguments for the purpose of showing them on the UI expect Unicode strings. Giving something else may result in garbled appearance of the text on the screen.

4.5 Date and Time

Unix time, seconds since January 1, 1970, 00:00:00 UTC (Coordinated Universal Time), is generally used as the time format in the Python for Series 60 APIs described in this document. The float type is used for storing time values.

4.6 Sharing Native Resources between Threads

Warning: Python for Series 60 objects that wrap native resources cannot be shared between threads. Trying this can lead to a crash.

In general, Python for Series 60 objects that wrap native resources cannot be shared between threads. Trying this can lead to a crash. This is because native resources cannot be shared between native threads. Two examples of this follow:

- Symbian OS STDLIB implementation has some limitations that are reflected at OS module support (see *Series 60 SDK documentation [4]*). For example, STDLIB file descriptors cannot be shared between threads, and for that reason, Python file objects cannot either.
- Sockets as implemented in the Series 60 version of the `socket` module have the same restriction.

4.7 Scalable User Interface

Note: Series 60 2nd Edition FP3 and further releases.

Series 60 2nd Edition FP3 enables a new feature called scalable user interface. For Python developers scalable user interface is currently visible in new APIs supporting the scalable UI, icon loading, and

new screen resolutions. For more information on scalable user interface, see Section 5.6, *Icon Type* of this document, as well as *Programming with Python for Series 60 Platform* [6].

4.8 Error Handling

The APIs described in this document may raise any standard Python exceptions. In situations where a Symbian error code is returned, its symbolic name is given as the value parameter of a `SymbianError` exception.

In case where the functions have nothing special to return, they return `None` on success.

4.9 Limitations and Areas of Development

Some OS level concepts to which the standard `os` library module offers an interface do not exist as such in Symbian OS environment. An example of this is the concept of *current working directory*.

Reference cycle garbage collection is not in use. Because of this, special care needs to be taken to dismantle cyclic references when a Python program exits. This prevents error messages related to native resources that are left open. The problem could be removed by developing support for collection of cyclic garbage or by performing a special cleanup action on interpreter exit. The `gc` module has been ported to the Symbian OS, and it has been verified to work. However, the current distribution has been built without `gc` support.

5 appuifw Module

The `appuifw` module offers an interface to Series 60 UI application framework. The services of this interface may only be used in the context of the main thread, that is, the initial thread of a UI application script.

5.1 Module Level Functions

The following free functions – functions that do not belong to any class – are defined in the `appuifw` module:

`available_fonts()`
Retrieves a list (Unicode) of all fonts available in the device.
Example:
```
deviceFonts = appuifw.available_fonts()
```

`input` `query(`*`label, type [,initial_value]`*`)`
Performs a query with a single-field dialog. The prompt is set to `label`, and the type of the dialog is defined by `type`. The value of `type` can be any of the following strings:

- `'text'`
- `'code'`
- `'number'`
- `'date'`
- `'time'`
- `'query'`

The type of the optional `initial_value` parameter and the returned input depend on the value of `type`:

- For text fields, (`'text'`, `'code'`) it is Unicode
- For number fields, it is numeric
- For date fields, it is seconds since epoch rounded down to the nearest local midnight

A simple confirmation query and time query take no initial value and return `True/None` and seconds since local midnight, correspondingly. All queries return `None` if the users cancel the dialog.

`(`*`input_1, input_2`*`) multi_query(`*`label_1, label_2`*`)`
A two-field text (Unicode) input dialog. Returns `None` if the users cancel the dialog.

`note(`*`text [, type]`*`)`
Displays a note dialog of the chosen type with *`text`* (Unicode). The default value for `type` is `'info'`, which is automatically used if `type` is not set. `type` can be one of the following strings: `'error'`, `'info'`, or `'conf'`.

`index` `popup_menu(`*`list [, label]`*`)`
A pop-up menu style dialog. *`list`* representing the menu contents can be a list of Unicode strings or a list of Unicode string pairs (tuples). The resulting dialog list is then a single-style or a double-style list. A single-style list is shown in full; whereas a double-style list shows the items one at a time. Returns `None` if the users cancel the operation.

index `selection_list(choices=`*list* `[, search_field=0])`

Executes a dialog that allows the users to select a list item and returns the *index* of the chosen item, or `None` if the selection is cancelled by the users.

`(choices=`*list* `[, search_field=`*0*`])` consists of keywords and values, where

- `choices` is a list of Unicode strings.
- `search_field` is `0` (disabled) by default and is optional. Setting it to `1` enables a search field (find pane) that facilitates searching for items in long lists. If enabled, the search field appears after you press a letter key.

(*indexes*) `multi_selection_list(choices=`*list* `[, style=`*'checkbox'*`, search_field=0])`

Executes a dialog that allows the users to select multiple list items. Returns a tuple of indexes (a pair of Unicode strings) of the chosen items, or `None` if the selection is cancelled by the users.

`(choices=`*list* `[, style=`*'checkmark'*`, search_field=0])` consists of keywords and values, where

- `choices` is a list of Unicode strings.
- `style` is an optional string; the default value being `'checkbox'`. If `'checkbox'` is given, the list will be a checkbox list, where empty checkboxes indicate what items can be marked. The other possible value that can be set for `style` is `'checkmark'`. If `'checkmark'` is given, the list will be a markable list, which lists items but does not indicate specifically that items can be selected. To select items on a markable list, use the **Navigation** key to browse the list and the **Edit** key to select an item. For example views on checkbox and markable lists, see Figure 3.
- `search_field` is `0` (disabled) by default and is optional. Setting it to `1` enables a search field (find pane) that facilitates searching for items in long lists. If enabled, the search field is always visible with checkbox lists; with markable lists it appears by pressing a letter key.

Example:

```
tuple = appuifw.multi_selection_list(L, style='checkmark',
search_field=1)
```

Figure 3: Examples of a checkbox list (left) and a markable list (right)

5.2 Application Type

A single implicit instance of this type always exists when `appuifw` module is present and can be referred to with the name `app`. New instances cannot be created by a Python program.

Instances of `Application` type have the following attributes:

`body`
The UI control that is visible in the application's main window. Currently either `Text`, a `Listbox` object, `Canvas`, or `None`.

`exit_key_handler`
A callable object that is called when the user presses the **Exit** softkey. Setting `exit_key_handler` to `None` sets it back to the default value.

`menu`
This can be:

- a list of `(`*`title`*`, `*`callback`*`)` pairs, where each pair describes an item in the application's menu bar, or
- a list of `(title, ((`*`title`*`, `*`callback`*`),...))` pairs, where the tuple of Unicode strings `((`*`title`*`, `*`callback`*`),...)` creates a submenu.

`title` (Unicode) is the name of the item and *`callback`* the associated callable object. The maximum allowed number of items in a menu, or items in a submenu, or submenus in a menu is 30.

Example:

```
appuifw.app.menu = [(u"item 1", item1),
                    (u"Submenu 1", ((u"sub item 1", subitem1),
                                    (u"sub item 2", subitem2) ))
                   ]
```

`screen`
The screen area used by an application. See Figure 4 for example screens. The appearance of the application on the screen can be affected by setting one of the following values: `'normal'`, `'large'`, and `'full'`.
Examples:
`appuifw.app.screen='normal'` (a normal screen with title pane and softkeys)
`appuifw.app.screen='large'` (only softkeys visible)
`appuifw.app.screen='full'` (a full screen)

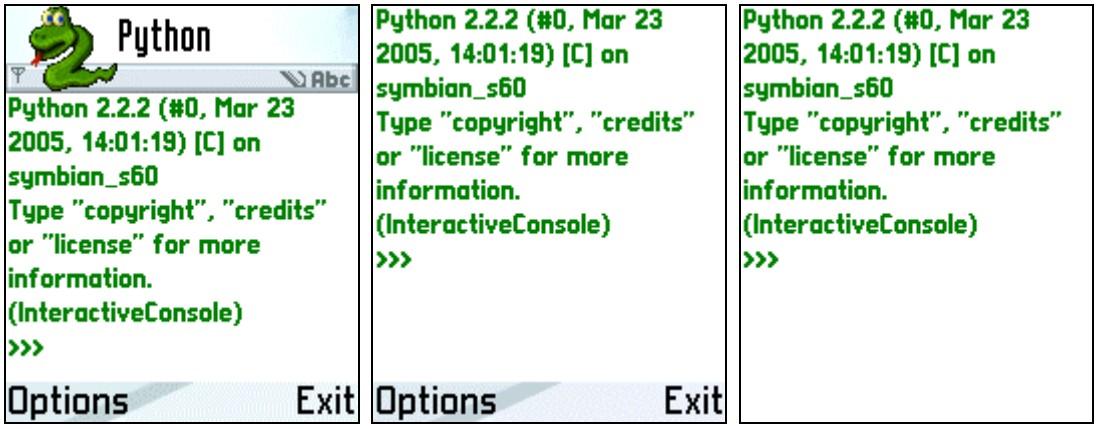

Figure 4: Examples of normal screen (left), large screen (middle), and full screen (right)

`title`
The title, in Unicode, that is visible in the application's title pane.

Instances of `Application` type have the following methods:

`activate_tab(`*`index`*`)`
Activates the tab *`index`* counting from zero.

`name` `full_name()`
Retrieves the full name, in Unicode, of the native application in whose context the current Python interpreter session runs.

`set_exit()`
Requests a graceful exit from the application as soon as the current script execution returns.

`set_tabs(`*`tab_texts`* `[,`*`callback=None`*`])`
Sets tabs with given names on them in the navigation bar; *`tab_texts`* is a list of Unicode strings. When the users navigate between tabs, *`callback`* gets called with the index of the active tab as an argument. Tabs can be disabled by giving an empty or one-item `tab_texts` list.

5.3 Form Type

`Form` implements a dynamically configurable, editable multi-field dialog. `Form` caters for advanced dialog use cases with requirements such as free selectability of the combination of fields, possibility of validating the user input, and automatically producing the contents of some dialog fields before allowing the closing of the dialog.

`Form([fields=`*`field_list`*`, flags=`*`flag`*`])`
Creates a `Form` instance. `field_list` consists of `tuples`: `(label, type [,value])`, where

- *`label`* is a Unicode string
- *`type`* is one of the following strings: *`'text'`*, *`'number'`*, *`'date'`*, *`'time'`*,or *`'combo'`*
- `value`, depending on `type`: Unicode string, numeric, float (seconds since Unix epoch rounded down to the nearest local midnight), float (seconds since local midnight), or `([unicode_string` *`choices`*`], `*`index`*`)`

`Form` can also be configured and populated after construction. The configuration flags are visible as an attribute. `Form` implements the list protocol that can be used for setting the form fields, as well as obtaining their values after the dialog has been executed.

Instances of `Form` type have the following attributes:

`flags`
This attribute holds the values of the various configuration flags. Currently supported flags are:

`FFormEditModeOnly`
When this flag is set, the form remains in edit mode while `execute` runs.

`FFormViewModeOnly`
When this flag is set, the form cannot be edited at all.

`FFormAutoLabelEdit`
This flag enables support for allowing the end-users to edit the labels of the form fields.

`FFormAutoFormEdit`
This flag enables automatic support for allowing the end-users to add and delete the form fields. Note that this is an experimental feature and is not guaranteed to work with all SDK versions.

`FFormDoubleSpaced`
When this flag is set, double-spaced layout is applied when the form is executed: one field takes two lines, as the label and the value field are on different lines.

`menu`
A list of `(title, callback)` pairs, where each pair describes an item in the form's menu bar that is active while the dialog is being executed. `title` (Unicode) is the name of the item and `callback` the associated callable object.

`save_hook`
This attribute can be set to a callable object that receives one argument and returns a Boolean value. It gets called every time the users want to save the contents of an executing `Form` dialog. A candidate list for new form content – a list representing the currently visible state of the UI – is given as an argument. The list can be modified by `save_hook`. If `save_hook` returns `True`, the candidate list is set as the new contents of the form. Otherwise, the form UI is reset to reflect the field list contained in `Form` object.

Instances of `Form` type have the following methods:

`execute()`
Executes the dialog by making it visible on the UI.

`insert()`, `pop()`, `length()`, subscription, subscription-assignment
A subset of the list interface to access the form instance as a list. The subscript notation `f[i]` can be used to access or modify the i-th element of the form `f`. Same limitations as discussed above in the context of the flag `FFormAutoFormEdit` apply to modifying a form while it is executing. The ability to change the schema of a form while it is executing is an experimental feature.

5.4 Text Type

`Text` is a text editor UI control. For examples on the options available with `Text`, see Figure 5.

Instances of `Text` type have the following attributes:

`color`
The color of the text. `color` supports the same color representation models as the `graphics` module. For the supported color representation models, see Section *6.4.3*, *Color representation*.

`focus`
A Boolean attribute that indicates the focus state of the control. Editor control also takes the ownership of the navigation bar, and this feature is needed to enable the usage of this control in applications that use the navigation bar – for example, navigation tabs.

`font`
The font of the text. There are two possible ways to set this attribute:

- Using a supported Unicode font, for example `u"Latin12"`. Trying to set a font which is not supported by the device has no effect. A list of supported fonts can be retrieved by using `appuifw.available_fonts`.

  Example, setting font:
  ```
  t = appuifw.Text()
  ```

```
t.font = u"albi17b"        (sets font to Albi 17 bold)
t.font = u"LatinPlain12"   (sets font to Latin Plain 12)
```

- Using one of the default device fonts that are associated with the following labels (plain strings):

```
'annotation'
'title'
'legend'
'symbol'
'dense'
'normal'
```

Example, setting font:

```
t.font = "title"   (sets font to the one used in titles)
```

Example, checking the currently set font:

```
unicodeFont = t.font
```

The attribute value retrieved is always a Unicode string. If the font has been set with a label, for example, `'title'`, the attribute will retrieve the font associated with that label.

`highlight_color`

The highlight color of the text. `highlight_color` supports the same color representation models as the `graphics` module. For the supported color representation models, see Section *6.4.3*, *Color representation*.

`style`

The style of the text. The flags for this attribute are defined in the `appuifw` module. These flags can be combined by using the binary operator `or`. The flags can be divided into two types: text style and text highlight. Text style flags can be freely combined with each other. However, one or more text style flags can be combined with only one text highlight flag. The flags are:

text style:

`STYLE_BOLD` enables bold text.
`STYLE_UNDERLINE` enables underlined text.
`STYLE_ITALIC` enables italicized text.
`STYLE_STRIKETHROUGH` enables marking text with strikethrough formatting.

text highlight:

`HIGHLIGHT_STANDARD` enables standard highlight.
`HIGHLIGHT_ROUNDED` enables rounded highlight.
`HIGHLIGHT_SHADOW` enables shadow highlight.

Note: Only one highlight is allowed to be used at once. Therefore, it is possible to combine only **one** highlight with one or more text styles.

Examples:

```
t = appuifw.Text()
```

These and other similar values and combinations are valid:

```
t.style = appuifw.STYLE_BOLD
t.style = appuifw.STYLE_UNDERLINE
t.style = appuifw.STYLE_ITALIC
t.style = appuifw.STYLE_STRIKETHROUGH
t.style =
(appuifw.STYLE_BOLD|appuifw.STYLE_ITALIC|appuifw.STYLE_UNDERLINE)
```

These values are valid:

```
t.style = appuifw.HIGHLIGHT_STANDARD
t.style = appuifw.HIGHLIGHT_ROUNDED
t.style = appuifw.HIGHLIGHT_SHADOW
```

This combination is **not** valid:

Note: Invalid code, do not try!

```
t.style = (appuifw.HIGHLIGHT_SHADOW|appuifw.HIGHLIGHT_ROUNDED)
```

Figure 5: Examples of the options available for `Text` type

Instances of `Text` type have the following methods:

`add(`*`text`*`)`
: Inserts the Unicode string `text` to the current cursor position.

`bind(`*`event_code, callback`*`)`
: Binds the callable Python object *`callback`* to event *`event_code`*. The key codes are defined in the *`key_codes`* library module. The call `bind(`*`event_code`*`, None)` clears an existing binding. In the current implementation the event is always passed also to the underlying native UI control.

`clear()`
: Clears the editor.

`delete([pos=0, len=len()])`
: Deletes `len` characters of the text held by the editor control, starting from the position `pos`.

`cursor_pos` `get_pos()`
: Returns the current cursor position.

`text_length` `len()`
: Returns the length of the text string held by the editor control.

`text` `get([pos=0, len=len()])`
: Retrieves `len` characters of the text held by the editor control, starting from the position `pos`.

`set(`*`text`*`)`
: Sets the text content of the editor control to Unicode string *`text`*.

`set_pos(`*`cursor_pos`*`)`
: Sets the cursor to *`cursor_pos`*.

5.5 Listbox Type

An instance of this UI control type is visible as a listbox, also known as a list in Symbian, that can be configured to be a single-line item or a double-item listbox. Figure 6 shows a single-line item Listbox with icons. For more information on the MBM and MIF formats, see Section *5.6*, *Icon Type*.

Figure 6: Listbox with icons

`Listbox(`*`list, callback`*`)`

Creates a `Listbox` instance. A callable object *`callback`* gets called when a listbox selection has been made. `list` defines the content of the listbox and can be one of the following:

- A normal (single-line item) listbox: a list of Unicode strings, for example `[unicode_string item1, unicode_string item2]`
- A double-item listbox: a two-element tuple of Unicode strings , for example `[(unicode_string item1, unicode_string item1description), (unicode_string item2, unicode_string item2description)]`
- A normal (single-line item) listbox with graphics: a two-element tuple consisting of a Unicode string and an `Icon` object, for example `[(unicode_string item1, icon1), (unicode_string item2, icon2)]`.
- A double-item listbox with graphics: a three-element tuple consisting of two Unicode strings and one `Icon` object, for example `[(unicode_string item1, unicode_string item1description, icon1), (unicode_string item2, unicode_string item2description, icon2)]`

Example:
To produce a normal (single-line item) lisbox with graphics:

```
icon1 = appuifw.Icon(u"z:\\system\\data\\avkon.mbm", 28, 29)
icon2 = appuifw.Icon(u"z:\\system\\data\\avkon.mbm ", 40, 41)

entries = [(u"Signal", icon1),
           (u"Battery", icon2)]

lb = appuifw.Listbox(entries, lbox_observe)
```

Instances of `Listbox` type have the following methods:

`bind(event_code, callback)`
Binds the callable Python object *`callback`* to event *`event_code`*. The key codes are defined in the `key_codes` library module. The call `bind(event_code, None)` clears an existing binding. In the current implementation the event is always passed also to the underlying native UI control.

`index` `current()`
Returns the currently selected item's index in the `Listbox`.

`set_list(`*`list`* `[,`*`current`*`])`
Sets the `Listbox` content to a list of Unicode strings or a list of tuples of Unicode strings. `list` can be one of the following:

- A normal (single-style) list: a list of Unicode strings, for example `[unicode_string item1, unicode_string item2]`
- A double-style list box: a two-element tuple of Unicode strings , for example `[(unicode_string item1, unicode_string item1description), (unicode_string item2, unicode_string item2description)]`
- A normal (single-style) list with graphics: a two-element tuple consisting of a Unicode string and an `Icon` object, for example `[(unicode_string item1, icon1), (unicode_string item2, icon2)]`.
- A double-style list box with graphics: a three-element tuple consisting of two Unicode strings and one `Icon` object, for example `[(unicode_string item1, unicode_string item1description, icon1), (unicode_string item2, unicode_string item2description, icon2)]`

Optionally, the initially focused list item can be set.

5.6 Icon Type

An instance of `Icon` type encapsulates an icon to be used together with a `Listbox` instance. Note that currently `Icon` can only be used with `Listbox` (see Section *5.5, Listbox Type*).

MBM is the native Symbian OS format used for pictures. It is a compressed file format where the files can contain several bitmaps and can be referred to by a number. An `.mbg` file is the header file usually associated with an `.mbm` file, which includes symbolic definitions for each bitmap in the file. For example, an `avkon.mbm` file has an associated index file called `avkon.mbg`, which is included in Series 60 SDKs. For more information on the MBM format and the bitmap converter tool, see *Series 60 SDK documentation [4]* and search the topics with the key term "How to provide Icons"; this topic also points you to the Bitmap Converter tool that can be used for converting bitmaps into the MBM format.

Note: Series 60 2nd Edition FP3 introduces a new format for icons called Multi-Image File (MIF). This format is very similar to the MBM format and also contains several compressed files. The files to be compressed should be in Scalable Vector Graphics Tiny (SVG-T) format. For more information on the SVG format, see *Scalable Vector Graphics (SVG) 1.1 Specification [10]*.

`Icon(`*`filename, bitmap, bitmapMask`*`)`
Creates an icon. *`filename`* is a Unicode file name and must include the whole path. Note that MBM and MIF (MIF only in Series 60 2nd Edition FP3) are the only file formats supported. *`bitmap`* and *`bitmapMask`* are integers that represent the index of the icon and icon mask inside that file respectively.

Example:
The following builds an icon with the standard signal symbol:

```
icon = appuifw.Icon(u"z:\\system\\data\\avkon.mbm", 28, 29)
```

5.7 Content_handler Type

An instance of `Content_handler` handles data content by its MIME type.

`Content_handler([callback])`
Creates a `Content_handler` instance. `Content_handler` handles data content by its MIME type. The optional *`callback`* is called when the embedded handler application started with the `open` method finishes.

Instances of `Content_handler` type have the following methods:

`open(filename)`
Opens the file *`filename`* (Unicode) in its handler application if one has been registered for the particular MIME type. The handler application is embedded in the caller's thread. The call to this function returns immediately. When the handler application finishes, the *`callback`* that was given to the `Content_handler` constructor is called.

`open_standalone(filename)`
Opens the file *`filename`* (Unicode) in its handler application if one has been registered for the particular MIME type. The handler application is started in its own process. The call to this function returns immediately. Note that *`callback`* is *not* called for applications started with this method.

5.8 Canvas Type

`Canvas` is a UI control that provides a drawable area on the screen and support for handling raw key events. `Canvas` supports the standard drawing methods that are documented in Chapter *6, graphics Module*.

`Canvas([redraw_callback=None, event_callback=None])`
Constructs a `Canvas`. The optional parameters are callbacks that are called when specific events occur. `redraw_callback` is called whenever a part of the `Canvas` has been obscured by something, is then revealed, and needs to be redrawn. This can typically happen, for example, when the user switches away from the Python application and back again, or after displaying a pop-up menu. The callback takes as its argument a four-element tuple that contains the top-left and the bottom-right corner of the area that needs to be redrawn. In many cases redrawing the whole `Canvas` is a reasonable option.

Note: Be careful of cyclic references here. For example, if the callbacks are methods of an object that holds a reference to the `Canvas`, a reference cycle is formed that must be broken at cleanup time or the `Canvas` will not be freed.

If this parameter is given and not set to `None`, `event_callback` is called whenever a raw key event is received. There are three kinds of key events: `EEventKeyDown`, `EEventKey`, and `EEventKeyUp`. When a user presses a key down, events `EEventKeyDown` and `EEventKey` are generated. When the key is released, an `EEventKeyUp` event is generated.

The argument to the `event_callback` is a dictionary that contains the following data for key events:

- `'type'`: one of `EEventKeyDown`, `EEventKey`, or `EEventKeyUp`
- `'keycode'`: the keycode of the key

- `'scancode'`: the scancode of the key
- `'modifiers'`: the modifiers that apply to this key event

Each key on the keyboard has one or more scancodes and zero or more keycodes associated with it. A scancode represents the physical key itself and a keycode is the result of state-related operating system defined processing done on the key. For keys that correspond to a symbol in the current character set of the phone, the keycode is equal to the code of the corresponding symbol in that character set. For example, if you are using the Nokia Wireless Keyboard (SU-8W), pressing the key **A** will always produce the scancode `65` (ASCII code for an upper case A), but the keycode could be either 65 or 91 (ASCII code for a lower case A) depending on whether or not the **Shift** key is pressed or **Caps Lock** is active.

The `key_codes` module contains definitions for the keycodes and scancodes. See Figure 7 for the codes of the most common keys on the phone keypad.

Some keys are handled in a special way:

- A short press of the **Edit** key causes it to stay down, meaning that no `EEventKeyUp` event is sent. The event is only sent after a long press.
- Detecting presses of the **Voice tags** key or the **Power** key is not supported.
- If the right softkey is pressed, the `appuifw.app.exit_key_handler` callback is always executed.

There is no way to prevent the standard action of the **Hang-up** key, the **Menu** key, the **Power** key or the **Voice tags** key from taking place.

1. EKeyLeftSoftkey
   EScancodeLeftSoftkey
2. EKeyYes
   EScancodeYes
3. EKeyMenu
   EScancodeMenu
4. EKey1...9,0
   EScancode1...9,0
5. EKeyStar
   EScancodeStar
6. EKeyLeftArrow
   EScancodeLeftArrow
7. EKeyUpArrow
   EScancodeUpArrow
8. EKeySelect
   EScancodeSelect

9. EKeyRightArrow
   EScancodeRightArrow
10. EKeyDownArrow
    EScancodeDownArrow
11. EKeyRightSoftkey
    EScancodeRightSoftkey
12. EKeyNo
    EScancodeNo
13. EKeyBackspace
    EScancodeBackspace
14. EKeyEdit
    EScancodeEdit
15. EKeyHash
    EScancodeHash

Figure 7: Keycodes and scancodes for phone keys usable from Python applications

Instances of `Canvas` type have the following attribute:

`size`
A two-element tuple that contains the current width and height of the `Canvas` as integers.

Instances of `Canvas` type have the same standard drawing methods that are documented in Chapter *6, graphics Module*.

6 graphics Module

The `graphics` module provides access to the graphics primitives and image loading, saving, resizing, and transformation capabilities provided by the Symbian OS.

The module is usable from both graphical Python applications and background Python processes. However, background processes have some restrictions. This means that plain string symbolic font names are not supported in background processes since background processes have no access to the UI framework (see also Section *6.4.4*, *Font specifications*).

For an example on using this module, see *Programming with Python for Series 60 Platform [6]*.

Note: Functions `Image.open` and `Image.inspect` and `Image` object methods `load`, `save`, `resize`, and `transpose` are not available for Series 60 1st Edition.

6.1 Module Level Functions

The following free functions – functions that do not belong to any class – are defined in the `graphics` module:

Image `screenshot()`
Takes a screen shot and returns the image in `Image` format.

6.2 Image Class Static Methods

The following `Image` class static methods are defined in the `graphics` module:

Image `Image.new(`*size*`[,mode='RGB16'])`
Creates and returns a new `Image` object with the given size and mode. `size` is a two-element tuple. `mode` specifies the color mode of the `Image` to be created. It can be one of the following:

- `'1'`: Black and white (1 bit per pixel)
- `'L'`: 256 gray shades (8 bits per pixel)
- `'RGB12'`: 4096 colors (12 bits per pixel)
- `'RGB16'`: 65536 colors (16 bits per pixel)
- `'RGB'`: 16.7 million colors (24 bits per pixel)

Image `Image.open(`*filename*`)` **Not supported in 1st Edition!**
Returns a new `Image` object (mode `RGB16`) that contains the contents of the named file. The supported file formats are JPEG and PNG. The file format is automatically detected based on file contents. *filename* should be a full path name.

dict `Image.inspect(`*filename*`)` **Not supported in 1st Edition!**
Examines the given file and returns a dictionary of the attributes of the file. At present the dictionary contains only the image size in pixels as a two-element tuple, indexed by key `'size'`. *filename* should be a full path name.

6.3 Image Objects

An `Image` object encapsulates an in-memory bitmap.

Note on asynchronous methods: Methods `resize`, `transpose`, `save`, and `load` have an optional callback argument. If the callback is not given, the method call is synchronous; when the method returns, the operation is complete or an exception has been raised. If the callback is given, the method calls are asynchronous. If all parameters are valid and the operation can start, the method call will return immediately. The actual computation then proceeds in the background. When it is finished, the callback is called with an error code as the argument. If the given code is `0`, the operation completed without errors, otherwise an error occurred.

It is legal to use an unfinished image as a source in a blit operation; this will use the image data as it is at the moment the blit is made and may thus show an incomplete result.

`Image` objects have the following methods:

`resize(`*`newsize`*`[,callback=`*`None`*`,keepaspect=0])` **Not supported in 1st Edition!**
Returns a new image that contains a resized copy of this image. If `keepaspect` is set to `1`, the resize will maintain the aspect ratio of the image, otherwise the new image will be exactly the given size.

If `callback` is given, the operation is asynchronous, and the returned image will be only partially complete until `callback` is called.

`transpose(`*`direction`*`[,callback=cb])` **Not supported in 1st Edition!**
Creates a new image that contains a transformed copy of this image. The `direction` parameter can be one of the following:

- `FLIP_LEFT_RIGHT`: Flips the image horizontally, exchanging left and right edges.
- `FLIP_TOP_BOTTOM`: Flips the image vertically, exchanging top and bottom edges.
- `ROTATE_90`: Rotates the image 90 degrees counterclockwise.
- `ROTATE_180`: Rotates the image 180 degrees.
- `ROTATE_270`: Rotates the image 270 degrees counterclockwise.

If `callback` is given, the operation is asynchronous and the returned image will be only partially complete until `callback` is called.

`load(`*`filename`*`[,callback=`*`None`*`])` **Not supported in 1st Edition!**
Replaces the contents of this `Image` with the contents of the named file, while keeping the current image mode. This `Image` object must be of the same size as the file to be loaded.

If `callback` is given, the operation is asynchronous and the loaded image will be only partially complete until `callback` is called. *`filename`* should be a full path name.

`save(`*`filename`*`[,callback=`*`None`*`, format=`*`None`*`, quality=75, bpp=24, compression=`*`'default'`*`])` **Not supported in 1st Edition!**
Saves the image into the given file. The supported formats are JPEG and PNG. If `format` is not given or is set to `None`, the format is determined based on the file name extension: '`.jpg`' or '`.jpeg`' are interpreted to be in JPEG format and '`.png`' to be in PNG format. *`filename`* should be a full path name.

When saving in JPEG format, the `quality` argument specifies the quality to be used and can range from 1 to 100.

When saving in PNG format, the `bpp` argument specifies how many bits per pixel the resulting file should have, and `compression` specifies the compression level to be used.
Valid values for `bpp` are:

- `1`: Black and white, 1 bit per pixel
- `8`: 256 gray shades, 8 bits per pixel
- `24`: 16.7 million colors, 24 bits per pixel

Valid values for `compression` are:

- `'best'`: The highest possible compression ratio, the slowest speed
- `'fast'`: The fastest possible saving, moderate compression
- `'no'`: No compression, very large file size
- `'default'`: Default compression, a compromise between file size and speed

If `callback` is given, the operation is asynchronous. When the saving is complete, the `callback` is called with the result code.

`stop()`
Stops the current asynchronous operation, if any. If an asynchronous call is not in progress, this method has no effect.

`Image` objects have the following attribute:

`size`
A two-element tuple that contains the size of the `Image`. Read-only.

6.4 Common Features of Drawable Objects

Objects that represent a surface that can be drawn on support a set of common drawing methods, described in this section. At present there are two such objects: `Canvas` from the `appuifw` module and `Image` from the `graphics` module.

6.4.1 Options

Many of these methods support a set of standard options. This set of options is as follows:

- `outline`: The color to be used for drawing outlines of primitives and text. If `None`, the outlines of primitives are not drawn.
- `fill`: The color to be used for filling the insides of primitives. If `None`, the insides of primitives are not drawn. If `pattern` is also specified, `fill` specifies the color to be used for areas where the pattern is white.
- `width`: The line width to be used for drawing the outlines of primitives.
- `pattern`: Specifies the pattern to be used for filling the insides of primitives. If given, this must be either `None` or a 1-bit (black and white) `Image`.

6.4.2 Coordinate representation

The methods accept an ordered set of coordinates in the form of a coordinate sequence. Coordinates can be of type `int`, `long`, or `float`. A valid coordinate sequence is a non-empty sequence of either

- Alternating x and y coordinates. In this case the sequence length must be even, or
- Sequences of two elements, that specify x and y coordinates.

Examples of valid coordinate sequences:

`(1, 221L, 3, 4, 5.85, -3)`: A sequence of three coordinates

`[(1,221L),(3,4),[5.12,6])`: A sequence of three coordinates

`(1,5)`: A sequence of one coordinate

`[(1,5)]`: A sequence of one coordinate

`[[1,5]]`: A sequence of one coordinate

Examples of **invalid** coordinate sequences:

Note: Invalid code, do not use!

`[]`: An empty sequence

`(1,2,3)`: Odd number of elements in a flat sequence

`[(1,2),(3,4),None]`: Contains an invalid element

`([1,2],3,4)`: Mixing the flat and nested form is not allowed

6.4.3 Color representation

All methods that take color arguments accept the following two color representations:

- A three-element tuple of integers in the range from 0 to 255 inclusive, representing the red, green, and blue components of the color.
- An integer of the form `0xrrggbb`, where `rr` is the red, `gg` the green, and `bb` the blue component of the color.

For 12 and 16 bit color modes the color component values are simply truncated to the lower bit depth. For the 8-bit grayscale mode images the color is converted into grayscale using the formula (2*r+5*g+b)/8, rounded down to the nearest integer. For 1-bit black and white mode images the color is converted into black (0) or white (1) using the formula (2*r+5*g+b)/1024.

Examples of valid colors:

`0xffff00`: Bright yellow

`0x004000`: Dark green

`(255,0,0)`: Bright red

`0`: Black

`255`: Bright blue

`(128,128,128)`: Medium gray

Examples of **invalid** colors:

Note: Invalid code, do not use!

`(0,0.5,0.9)`: Floats are not supported

`'#ff80c0'`: The HTML color format is not supported

`(-1,0,1000)`: Out-of-range values

`(1,2)`: The sequence is too short

`[128,128,192]`: This is not a tuple

6.4.4 Font specifications

`Font` can be specified in two ways. Either as a Unicode string that represents a full font name, such as `u'LatinBold19'`, or as a plain string symbolic name that refers to a font setting currently specified by the UI framework. You can obtain a list of all available fonts with the `appuifw` module function `available_fonts`.

The symbolic names for UI fonts are:

- `'normal'`
- `'dense'`
- `'title'`
- `'symbol'`
- `'legend'`
- `'annotation'`

Note: Since background processes have no access to the UI framework, these symbolic names are not supported in them. You need to specify the full font name.

6.4.5 Common Methods of Drawable Objects

`line(`*`coordseq`*`[,<`*`options`*`>])`
Draws a line connecting the points in the given coordinate sequence. For more information about the choices available for *`options`*, see Section *6.4.1*, *Options*.

`polygon(`*`coordseq`*`[,<`*`options`*`>])`
Draws a line connecting the points in the given coordinate sequence, and additionally draws an extra line connecting the first and the last point in the sequence. If a fill color or pattern is specified, the polygon is filled with that color or pattern. For more information about the choices available for *`options`*, see Section *6.4.1*, *Options*.

`rectangle(`*`coordseq`*`[,<`*`options`*`>])`
Draws rectangles between pairs of coordinates in the given sequence. The coordinates specify the top-left and the bottom-right corners of the rectangle. The sequence must have an even number of coordinates. For more information about the choices available for *`options`*, see Section *6.4.1*, *Options*.

`ellipse(coordseq[,<options>])`

`pieslice(coordseq, start, end, [,<options>])`

`arc(coordseq, start, end, [,<options>])`

Draws complete ellipses or portions of an ellipse between pairs of coordinates in the given sequence. The coordinates specify the top-left and bottom-right corners of the rectangle inside which the ellipse is contained. The sequence must have an even number of coordinates.

The `ellipse` method draws complete ellipses, the `pieslice` method draws sectors of the ellipse and the `arc` method draws portions of the ellipse arc. The start and end parameters are floats that specify the start and end points of the `pieslice` or `arc` as the starting and ending angle in radians. The angle `0` is to the right, the angle `pi/2` is straight up, `pi` is to the left and `-pi/2` is straight down.

For more information about the choices available for *`options`*, see Section *6.4.1*, *Options*.

`point(coordseq, start, end, [,<options>])`

Draws points in each coordinate in the given coordinate sequence. If the `width` option is set to greater than 1, draws a crude approximation of a circle filled with the outline color in the locations. Note that the approximation is not very accurate for large widths; use the `ellipse` method if you need a precisely formed circle. For more information about the choices available for *`options`*, see Section *6.4.1*, *Options*.

`clear([color=0xffffff])`

Sets the entire surface of the drawable to the given color, white by default.

`text(coordseq, text, [fill=0, font=u"LatinBold12"])`

Draws the given text in the points in the given coordinate sequence with the given color (default value is black) and the given font (default value is `u"LatinBold12"`). The font specification format is described above.

`blit(image[,target=(0,0),source=((0,0),image.size),mask=None,scale=0])`

Copies the source area from the given `image` to the target area in this drawable. The source area is copied in its entirety if `mask` is not given or is set to `None`. If the mask is given, the source area is copied where the mask is white. `mask` can be either `None` or a 1-bit (black and white) `Image` and must be of the same size as the source image.

`target` and `source` specify the target area in this image and the source area in the given source. They are coordinate sequences of one or two coordinates. If they specify one coordinate, it is interpreted as the upper-left corner for the area; if they specify two coordinates, they are interpreted as the top-left and bottom-right corners of the area.

If `scale` is other than zero, scaling is performed on the fly while copying the source area to the target area. If `scale` is zero, no scaling is performed, and the size of the copied area is clipped to the smaller of source and target areas.

Note that a `blit` operation with scaling is slower than one without scaling. If you need to blit the same `Image` many times in a scaled form, consider making a temporary `Image` of the scaling result and blitting it without scaling. Note also that the scaling performed by the `blit` operation is much faster but of worse quality than the one done by the `resize` method, since the `blit` method does not perform any antialiasing.

7 e32 Module

The `e32` module offers Symbian OS related utilities that are not related to the UI and are not provided by the standard Python library modules.

7.1 Module Level Functions

The following free functions – functions that do not belong to any class – are defined in the `e32` module:

`ao_yield()`
: Yields to the *active scheduler* to have ready active objects with priority above normal scheduled for running. This has the effect of flushing the eventual pending UI events. Note that the UI callback code may be run in the context of the thread that performs an `ao_yield`. For information on active scheduler, see *Series 60 SDK documentation [4]*.

`ao_sleep(`*`interval`* `[,`*`callback`*`])`
: Sleeps for the given *`interval`* without blocking the active scheduler. When the optional *`callback`* is given, the call to `ao_sleep` returns immediately and the *`callback`* gets called after *`interval`*.

`callgate` `ao_callgate(`*`wrapped_callable`*`)`
: Wraps *`wrapped_callable`* into a callable object *`callgate`* that can be called in any thread. As a result of a call to *`callgate`*, *`wrapped_callable`* gets called in the context of the thread that originally created the callgate. Arguments can be given to the call. This is actually a simple wrapping of the Symbian active object facility.

`driveletters` `drive_list()`
: Returns a list of currently visible drives as a list of Unicode strings *`'<driveletter>:'`*

`file_copy(`*`target_name, source_name`*`)`
: Copies the file *`source_name`* to *`target_name`*. The names must be complete paths.

`mode` `in_emulator()`
: Returns `1` if running in an emulator, or `0` if running on a device.

`pys60_version`
: A string containing the version number of the Python for Series 60 and some additional information.
Example:
```
>>> import e32
>>> e32.pys60_version
'1.2 final'
```

`pys60_version_info`
: A tuple containing the five components of the Python for Series 60 version number: major, minor, micro, release level, and serial. All values except release level are integers; the release level is a string and is either `'alpha'`, `'beta'`, `'candidate'`, or `'final'`. The `pys60_version_info` value corresponding to the Python for Series 60 version 1.2 is `(1, 2, 'final', 0)`.

`s60_version_info`
: The SDK version with which this Python was compiled (tuple). The following values are possible:

- `(1, 2)` for Series 60 1st Edition
- `(2, 0)` for Series 60 2nd Edition
- `(2, 6)` Series 60 2nd Edition Feature Pack 2

Examples:

```
>>> import e32
>>> e32.pys60_version
'1.2 final'
>>> e32.pys60_version_info
(1, 2, 'final', 0)
>>> e32.s60_version_info
(2, 0)
>>>
```

`bool is_ui_thread()`
: Returns `True` if the code that calls this function runs in the context of the UI thread; otherwise returns `False`.

`start_exe(`*`filename, command [,wait]`*`)`
: Launches the native Symbian OS executable *`filename`* (Unicode) and passes it the *`command`* string. When *`wait`* is set, the function synchronously waits for the exit of the executable and returns a value that describes the exit type. Possible values are `0` for normal exit and `2` for abnormal exit.

`start_server(`*`filename`*`)`
: Starts the Python script in file *`filename`* (Unicode) as a server in its own process. Note that `appuifw` module is not available to a server script.

7.2 Ao_lock Type

`Ao_lock()`
: Creates an `Ao_lock` instance. A Symbian active object based synchronization service. This can be used in the main thread without blocking the handling of UI events. The application should not exit while a thread is waiting in `Ao_lock`. If `Ao_lock` is called while another wait is in progress, an `AssertionError` is raised.

Instances of `Ao_lock` type have the following methods:

`wait()`
: If the lock has already been signaled, returns immediately. Otherwise blocks in wait for the lock to be signaled. Only one waiter is allowed, so you should avoid recursive calls to this service. `wait` can only be called in the thread that created the lock object. During the wait, other Symbian-active objects are being served, so the UI will not freeze. This may result in the UI callback code being run in the context of the thread that is waiting in `Ao_lock`. This must be considered when designing the application logic.

`signal()`
: Signals the lock. The waiter is released.

8 e32db Module

The `e32db` module provides an API for relational database manipulation with a restricted SQL syntax. For details of DBMS support, see *Series 60 SDK documentation [4]*.

8.1 Module Level Functions

The following free functions – functions that do not belong to any class – are defined in the `e32db` module:

`datetimestring` `format_rawtime(`*`timevalue`*`)`
Formats the given time value (Symbian time) according to the current system's date/time formatting rules and returns it as a Unicode string.

`datetimestring` `format_time(`*`timevalue`*`)`
Formats the given time value according to the current system's date/time formatting rules and returns it as a Unicode string.

8.2 Dbms Type

`Dbms()`
Creates a `Dbms` object. `Dbms` objects support basic operations on a database.

Instances of `Dbms` type have the following methods:

`begin()`
Begins a transaction on the database.

`close()`
Closes the database object. It is safe to try to close a database object even if it is not open.

`commit()`
Commits the current transaction.

`compact()`
Compacts the database, reclaiming unused space in the database file.

`create(`*`dbname`*`)`
Creates a database with path *`dbname`*.

`retval` `execute(`*`query`*`)`
Executes an SQL *`query`*. On success, returns 0 if a DDL (SQL schema update) statement was executed. Returns the number of rows inserted, updated, or deleted, if a DML (SQL data update) statement was executed.

`open(`*`dbname`*`)`
Opens the database in file *`dbname`*. This should be a full Unicode path name, for example, `u'c:\\foo.db'`.

`rollback()`
Rolls back the current transaction.

8.3 DB_view Type

`Db_view`
: Creates a `Db_view` object. `DB_view` objects generate rowsets from a SQL query. They provide functions to parse and evaluate the rowsets.

Instances of `Db_view` type have the following methods:

value `col(`*column*`)`
: Extracts a *value* from *column* and returns the corresponding Python value. The first column of the rowset has the index 1. If the type of the column is not supported, a `TypeError` is raised. See Table 1 for a list of supported data types.

cols `col_count()`
: Returns the number of columns defined in the rowset.

len `col_length(`*column*`)`
: Gets the length of the value in *column*: empty columns have a length of zero; non-empty numerical and date/time columns have a length of 1. For text columns, the length is the character count, and for binary columns, the length is the byte count.

value `col_raw(`*column*`)`
: Extracts the value of *column* as raw binary data, and returns it as a Python string. The first column of the rowset has the index 1. See Table 1 for a list of supported data types.

value `col_rawtime(`*column*`)`
: Extracts the value of date/time column as a long integer, which represents the raw Symbian time value. The first column of the rowset has the index 1. See Table 1 for a list of the supported data types.

value `col_type(`*column*`)`
: Returns the numeric type of the given column as an integer from a Symbian-specific list of types. This function is used in the implementation of method `col`.

rows `count_line()`
: Returns the number of rows available in the rowset.

`first_line()`
: Positions the cursor on the first row in the rowset.

`get_line()`
: Gets the current row data for access.

`is_col_null(`*column*`)`
: Tests whether *column* is empty. Empty columns can be accessed like normal columns. Empty numerical columns return a `0` or an equivalent value, and text and binary columns have a zero length.

`next_line()`
: Moves the cursor to the next row in the rowset.

`prepare(`*db*`, `*query*`)`
: Prepares the view object for evaluating an SQL select statement. *db* is a `Dbms` object and *query* the SQL query to be executed.

For examples on using this module, see *Programming with Python for Series 60 Platform [6]*.

8.4 Mapping Between SQL and Python Data Types

See Table 1 for a summary of mapping between SQL and Python data types. The `col` function can extract any value except `LONG VARBINARY` and return it as the proper Python value. In addition, the `col_raw` function can extract any column type except `LONG VARCHAR` and `LONG VARBINARY` as raw binary data and return it as a Python string.

Inserting, updating, or searching for `BINARY`, `VARBINARY`, or `LONG VARBINARY` values is not supported. `BINARY` and `VARBINARY` values can be read with `col` or `col_raw`.

8.5 Date and Time Handling

The functions `col` and `format_time` use Unix time, seconds since January 1, 1970, 00:00:00 UTC, as the time format. Internally the database uses the native Symbian time representation that provides greater precision and range than the Unix time. The native Symbian time format is a 64-bit value that represents microseconds since January 1st 0 AD 00:00:00 local time, nominal Gregorian. BC dates are represented by negative values. Since converting this format to Unix time and back may cause slight round-off errors, you have to use the functions `col_rawtime` and `format_rawtime` if you need to be able to handle these values with full precision.

The representation of date and time literals in SQL statements depends on the current system date and time format. Note that the only accepted ordering of day, month, and year is the one that the system is currently configured to use. Dates in other order are rejected. The recommended way to form date/time literals for SQL statements is to use the functions `format_time` or `format_rawtime` that format the given date/time values properly according to the current system's date/time format settings.

Table 1: Mapping between SQL and Python types

SQL type	Symbian column type (in the DBMS C++ API)	Python type	Supported
BIT	EDbColBit	int	yes
TINYINT	EDbColInt8		
UNSIGNED TINYINT	EDbColUint8		
SMALLINT	EDbColInt16		
UNSIGNED SMALLINT	EDbColUint16		
INTEGER	EDbColInt32		
UNSIGNED INTEGER	EDbColUint32		
COUNTER	EDbColUint32 (with the TDbCol::EAutoIncrement attribute)		
BIGINT	EDbColInt64	long	
REAL	EDbColReal32	float	
FLOAT	EDbColReal64		
DOUBLE			
DOUBLE PRECISION			
DATE	EDbColDateTime	float (or long, with col_rawtime())	
TIME			
TIMESTAMP			
CHAR(n)	EDbColText	Unicode	
VARCHAR(n)			
LONG VARCHAR	EDbColLongText		
BINARY(n)	EDbColBinary	str	read only
VARBINARY(n)			
LONG VARBINARY	EDbColLongBinary	n/a	no

9 e32dbm Module

The `e32dbm` module provides a DBM API that uses the native Symbian RDBMS as its storage back-end. The module API resembles that of the `gdbm` module. The main differences are:

- The `firstkey()` - `nextkey()` interface for iterating through keys is not supported. Use the `"for key in db"` idiom or the `keys` or `keysiter` methods instead.
- This module supports a more complete set of dictionary features than `gdbm`.
- The values are always stored as Unicode, and thus the values returned are Unicode strings even if they were given to the DBM as normal strings.

9.1 Module Level Functions

The following free functions – functions that do not belong to any class – are defined in the `e32dbm` module:

e32dbm_object `open(`*dbname[,flags[,mode]]*`)`

Opens or creates the given database file and returns *e32dbm_object*. Note that *dbname* should be a full path name, for example, `u'c:\\foo.db'`. Flags can be:

- '`r`' - opens an existing database in read-only mode. This is the default value.
- '`w`' - opens an existing database in read-write mode.
- '`c`' – opens a database in read-write mode. Creates a new database if the database does not exist.
- '`n`' – creates a new empty database and opens it in read-write mode.

If the character 'f' is appended to flags, the database is opened in *fast mode*. In fast mode, updates are written to the database only when one of these methods is called: `sync`, `close`, `reorganize`, or `clear`.

Since the connection object destructor calls `close`, it is not strictly necessary to close the database before exiting to ensure that data is saved, but it is still good practice to call the `close` method when you are done with using the database. Closing the database releases the lock on the file and allows the file to be reopened or deleted without exiting the interpreter.

Note: If you plan to do several updates, it is highly recommended that you open the database in fast mode, since inserts and updates are more efficient when they are bundled together in a larger transaction. This is especially important when you plan to insert large amounts of data, since inserting records to `e32db` is very slow if done one record at a time.

9.2 e32dbm Objects

The `e32dbm` objects returned by the `open` function support most of the standard dictionary methods. The supported dictionary methods are:

- `__getitem__`
- `__setitem__`
- `__delitem__`
- `has_key`
- `update`
- `__len__`

- `__iter__`
- `iterkeys`
- `iteritems`
- `itervalues`
- `get`
- `setdefault`
- `pop`
- `popitem`
- `clear`

These work the same way as the corresponding methods in a normal dictionary.

In addition, `e32dbm` objects have the following methods:

`close()`
: Closes the database. In fast mode, commits all pending updates to disk. `close` raises an exception if called on a database that is not open.

`reorganize()`
: Reorganizes the database. Reorganization calls `compact` on the underlying `e32db` database file, which reclaims unused space in the file. Reorganizing the database is recommended after several updates.

`sync()`
: In fast mode, commits all pending updates to disk.

10 messaging Module

The `messaging` module offers APIs to messaging services. Currently, the `messaging` module has one function:

`sms_send(`*`recipient`*`, `*`message`*`)`
: Sends an SMS message with body text *`message`* (Unicode) to telephone number *`recipient`* (string).

11 location Module

The `location` module offers APIs to location information related services. Currently, the `location` module has one function:

mcc, *mnc*, *lac*, *cellid* gsm_location()
: Retrieves GSM location information: Mobile Country Code, Mobile Network Code, Location Area Code, and Cell ID. A location area normally consists of several base stations. It is the area where the terminal can move without notifying the network about its exact position. *mcc* and *mnc* together form a unique identification number of the network into which the phone is logged.

12 sysinfo Module

The `sysinfo` module offers an API for checking the system information of a Series 60 mobile device.

Note: The method `ring_type` is not available for Series 60 1st Edition.

The `sysinfo` module has the following functions:

`battery` `battery()`
Returns the current battery level ranging from 0 to 7, with 0 meaning that the battery is empty and 7 meaning that the battery is full. If using an emulator, value 0 is always returned.

`(`*`width, height`*`) display_twips()`
Returns the width and height of the display in twips. For a definition of a twip, see Chapter *19*, *Terms and Abbreviations*.

`(`*`width, height`*`) display_pixels()`
Returns the width and height of the display in pixels.

`free_space` `free_drivespace()`
Returns the amount of free space left on the drives in bytes, for example `{u'C:' 100}`. The keys in the dictionary are the drive letters followed by a colon (:).

`imei` `imei()`
Returns the IMEI code of the device as a Unicode string. If using an emulator, the hardcoded string `u'000000000000000'` is returned.

`maxram` `max_ramdrive_size()`
Returns the maximum size of the RAM drive on the device.

`ram` `total_ram()`
Returns the amount of RAM memory on the device.

`free_ram` `free_ram()`
Returns the amount of free RAM memory available on the device.

`rom` `total_rom()`
Returns the amount of read-only ROM memory on the device.

`ringtype` `ring_type()` **Not supported in 1st Edition!**
Returns the current ringing type as a string, which can be one of the following: `'normal'`, `'ascending'`, `'ring_once'`, `'beep'`, or `'silent'`.

`(`*`major`*`,` *`minor`*`,` *`build`*`) os_version()`
Returns the operating system version number of the device as integers. The returned version is defined by a set of three numbers as follows[1]:

- The major version number, ranging from 0 to 127 inclusive
- The minor version number, ranging from 0 to 99 inclusive
- The build number, ranging from 0 to 32767 inclusive.

`signal` `signal()`
Returns the current network signal strength ranging from 0 to 7, with 0 meaning no signal and 7 meaning a strong signal. If using an emulator, value 0 is always returned.

[1] Descriptions for these values are based on information found in Series 60 SDK documentation [*4*].

sw_version sw_version()

Returns the software version as a Unicode string. If using an emulator, the hardcoded string `u'emulator'` is returned. For example, a software version can be returned as `u'V 4.09.1 26-02-04 NHL-10 (c) NMP'`.

13 camera Module

Note: Not available for Series 60 1st Edition.

The `camera` module enables taking photographs.

The `camera` module has the following functions[1]:

number `cameras_available()`
Returns the number of cameras available in the device.

values `image_modes()`
Returns the image modes supported in the device as a list of strings, for example: `['RGB12', 'RGB', 'RGB16']`.

values `image_sizes()`
Returns the image sizes (resolution) supported in the device as a list of `(x, y)` tuples, for example: `[(640, 480), (160, 120)]`.

modes `flash_modes()`
Returns the flash modes available in the device as a list of strings.

value `max_zoom()`
Returns the maximum digital zoom value supported in the device as an integer.

modes `exposure_modes()`
Returns the exposure settings supported in the device as a list of strings.

modes `white_balance_modes()`
Returns the white balance modes available in the device as a list of strings.

Image `take_photo([mode='RGB16', size=(640, 480), flash='auto', zoom=0, exposure='auto', white_balance='auto', position=0])`
Takes a photograph and returns the image in `Image` format (for more information on `Image` format, see Chapter *6, graphics Module*). If some other application is using the camera, this operation fails, for example with `SymbianError: KErrInUse`. The settings listed below describe all settings that are supported by the `camera` module. You can retrieve the mode settings available for your device by using the appropriate functions listed at the beginning of this chapter.

- `mode` is the display mode of the image. The default value is `'RGB16'`. The following display modes are supported:
  - `'RGB12'`: 4096 colors (12 bits per pixel)
  - `'RGB16'`: 65536 colors (16 bits per pixel). Default value, always supported
  - `'RGB'`: 16.7 million colors (24 bits per pixel)
- `size` is the resolution of the image. The default value is `(640, 480)`. The following sizes are supported, for example, in Nokia 6630: `(1280, 960)`, `(640, 480)` and `(160, 120)`.
- `flash` is the flash mode setting. The default value is `'none'`. The following flash mode settings are supported:
  - `'none'`
    No flash. Default value, always supported

[1] Descriptions for some of the values are based on information found in Series 60 SDK documentation [4].

- `'auto'`
  Flash will automatically fire when required
- `'forced'`
  Flash will always fire
- `'fill_in'`
  Reduced flash for general lighting
- `'red_eye_reduce'`
  Red-eye reduction mode

- `zoom` is the digital zoom factor. It is assumed to be on a linear scale from 0 to the maximum zoom value allowed in the device. The default value is `0`, meaning that zoom is not used.
- `exposure` is the exposure adjustment of the device. Exposure is a combination of lens aperture and shutter speed used in taking a photograph. The default value is `'auto'`. The following exposure modes are supported:
  - `'auto'`
    Sets exposure automatically. Default value, always supported
  - `'night'`
    Night-time setting for long exposures
  - `'backlight'`
    Backlight setting for bright backgrounds
  - `'center'`
    Centered mode for ignoring surroundings
- `white_balance` can be used to adjust white balance to match the main source of light. The term white balance refers to the color temperature of the current light. A digital camera requires a reference point to represent white. It will then calculate all the other colors based on this white point. The default value for `white_balance` is `'auto'` and the following white balance modes are supported:
  - `'auto'`
    Sets white balance automatically. Default value, always supported
  - `'daylight'`
    Sets white balance to normal daylight
  - `'cloudy'`
    Sets white balance to overcast daylight
  - `'tungsten'`
    Sets white balance to tungsten filament lighting
  - `'fluorescent'`
    Sets white balance to fluorescent tube lighting
  - `'flash'`
    Sets white balance to flash lighting
- `position` is the camera used if the device, such as Nokia 6680, has several cameras. In Nokia 6680, the camera pointing to the user of the device is located in position `1`, whereas the one pointing away from the user is located in position `0`. The default `position` is `0`.

14 audio Module

The `audio` module enables recording and playing audio files. The `audio` module supports all the formats supported by the device, typically: WAV, AMR, MIDI, MP3, AAC, and Real Audio[1]. For more information on the audio types supported by different devices, see the *Forum Nokia* Web site *[7]* and *Series 60 Platform* Web site [*8*].

14.1 Sound Class Static Methods

The following `Sound` class static methods are defined in the `audio` module:

Sound `Sound.open(`*`filename`*`)`
: Returns a new initialized `Sound` object with the named file opened. Note that *`filename`* should be a full Unicode path name and must also include the file extension, for example `u'c:\\foo.wav'`.

14.2 Sound Objects

`Sound` objects have the following functions:

`play([times=1, interval=0])`
: Starts playback of an audio file from the beginning. Without the parameters `times` and `interval` it plays the audio file one time. `times` defines the number of times the audio file is played, the default being `1`. If the audio file is played several times, `interval` gives the time interval between the subsequent plays in microseconds. Other issues:

- Calling `play(audio.KMdaRepeatForever)` will repeat the file forever.
- If an audio file is played but not stopped before exiting, the Python script will leave audio playing on; therefore `stop` needs to be called explicitly prior to exit.
- Currently the module does not support playing simultaneous audio files, calling `play` to a second `Sound` instance while another audio file is playing, stops the earlier audio file and starts to play the second `Sound` instance.
- Calling `play` while a telephone call is ongoing plays the sound file to uplink. In some devices the sound file is also played to the device speaker.
- Calling `play` when already playing or recording results in `RuntimeError`. Calling `stop` prior to `play` will prevent this from happening.

`stop()`
: Stops playback or recording of an audio file.

`record()`
: Starts recording audio data to a file. If the file already exists, the operation appends to the file. For Nokia devices, WAV is typically supported for recording. For more information on the audio types supported by different devices, see the *Forum Nokia* Web site [*7*] and *Series 60 Platform* Web site [*8*]. Other issues:

- Calling `record` while a telephone call is ongoing starts the recording of the telephone call.
- Calling `record` when already playing or recording results in `RuntimeError`. Calling `stop` prior to `record` will prevent this from happening.

[1] The dynamically loaded audio codec for the sound file is based on the MIME-type information inside the audio file and file extension.

`close()`
Closes an opened audio file.

`state` `state()`
Returns the current state of the `Sound` type instance. The different states (constants) are defined in the `audio` module. The possible states[1] are:

- `ENotReady`
  The `Sound` object has been constructed but no audio file is open.
- `EOpen`
  An audio file is open but no playing or recording operation is in progress.
- `EPlaying`
  An audio file is playing.
- `ERecording`
  An audio file is being recorded.

[1] Descriptions for these options are based on information found in Series 60 SDK documentation [4].

15 telephone Module

This module provides an API to a telephone.

Since the users of the device can also hang-up the phone explicitly, they might affect the current status of the call. In addition, using this extension in an emulator has no effect since no calls can be connected.

The `telephone` module has the following functions:

`dial(`*`number`*`)`
: Dials the number set in *`number`*. *`number`* is a string, for example u'`+358501234567`' where '`+`' is the international prefix, '`358`' is the country code, '`50`' is the mobile network code (or the area code), and '`1234567`' is the subscriber number. If there is an ongoing phone call prior to calling `dial` from Python, then the earlier call is put on hold and a new call is established. Calling `dial` multiple times when, for example, the first call has been answered and a line has been established results in subsequent calls not being connected.

`hang_up()`
: Hangs up if a call initiated by `dial` is in process. If this call has already been finished, `SymbianError: KErrNotReady` is raised.

16 calendar Module

The `calendar` module offers an API to calendar services. The `calendar` module represents a Symbian agenda database as a dictionary-like `CalendarDb` object, which contains `Entry` objects and which is indexed using the unique IDs of those objects. There are four types of entry objects: `AppointmentEntry`, `EventEntry`, `AnniversaryEntry`, and `TodoEntry`.

`CalendarDb` objects represent a live view into the database. If an entry is changed outside your Python application, the changes are visible immediately, and conversely any changes you commit into the database are visible immediately to other applications.

In addition to entries, there are todo lists which contain todo entries. Todo lists are accessed using the dictionary-like `TodoListDict` and `TodoList` objects.

All time parameters use Unix time unless stated otherwise. For more information on Unix time, see Section *4.5*, *Date and Time*.

Figure 8 demonstrates the relationships of the `calendar` module objects.

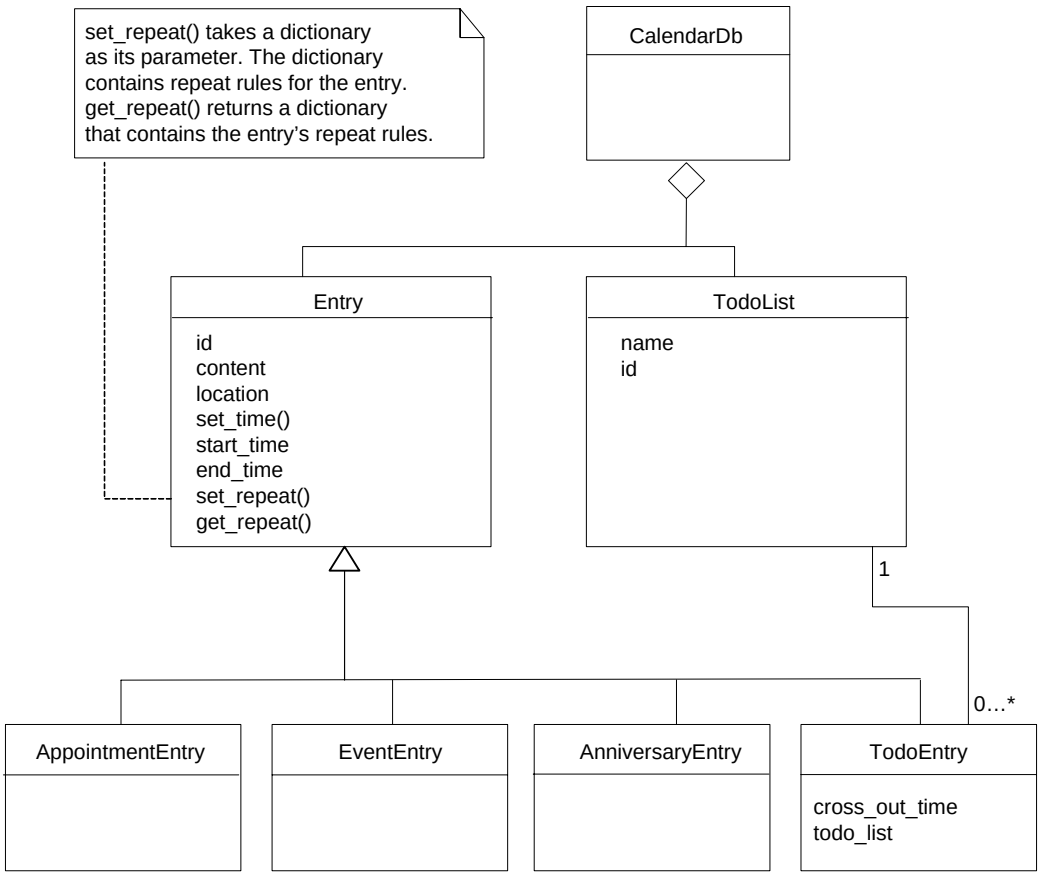

Figure 8: The `calendar` module objects

16.1 Module Level Functions

The following free functions – functions that do not belong to any class – are defined in the `calendar` module:

Calendardb `open()`
: Opens the default `calendar` database. Returns a new `CalendarDb` object.

Calendardb `open(`*filename*`)`
: Opens the specified `calendar` database. Returns a new `CalendarDb` object. *filename* should be a full Unicode path name.

Calendardb `open(`*filename*`, 'c')`
: Opens the specified `calendar` database. If it does not already exist, a new one is created. Returns a new `CalendarDb` object. *filename* should be a full Unicode path name.

Calendardb `open(`*filename*`, 'n')`
: Creates a new, empty `calendar` database. *filename* should be a full Unicode path name. Setting `filename` to an existing file erases the old file and creates a new file by the same name. Returns a new `CalendarDb` object.

16.2 CalendarDb Objects

Calendar entries and todo lists are stored in a calendar database. There is one default calendar database but more calendar databases can be created by invoking `open` with parameters `'n'` or `'c'`.

`CalendarDb` objects have the following methods:

AppointmentEntry `add_appointment()`
: Creates a new appointment entry. The entry is not added and saved into the database until `Entry.commit` is called.

EventEntry `add_event()`
: Creates a new event entry. The entry is not added and saved into the database until `Entry.commit` is called.

AnniversaryEntry `add_anniversary()`
: Creates a new anniversary entry. The entry is not added and saved into the database until `Entry.commit` is called.

TodoEntry `add_todo()`
: Creates a new todo entry. The entry is not added and saved into the database until `Entry.commit` is called.

entry_instances
`find_instances(`*start_date*`,`*end_date*`,search_str=u''[,appointments=0,events=0,anniversaries=0,todos=0])`
: The parameters for this function include the start date, end date, search string, and optional parameters. The optional parameters define the entry types to be included into the search. By default all entry types are included. Returns a list that contains `Entry` instances found in the search. An instance is a dictionary that contains the entry ID and the datetime value. An entry may have several instances if it is repeated, for example once every week, etc. However, all the returned instances occur on the same day, i.e. on the first day between the start and end datetime values that contains instances. To search all instances between the initial start and end datetime values, you may have to execute several searches and change the start datetime value for each search. A match is detected if the search string is a substring of an entry's content.

entry_instances
monthly_instances(*month*,appointments=0,events=0,anniversaries=0,todos=0)
The parameters for this function include *month* (float) and optional parameters. The optional parameters define the entry types to be returned. Returns a list that contains entry instances occurring during the specified calendar month.

entry_instances
daily_instances(*day*,appointments=0,events=0,anniversaries=0,todos=0)
The parameters for this function include *day* (float) and optional parameters. The optional parameters define the entry types to be returned. Returns a list that contains entry instances occurring on the specified day.

id add_todo_list([name=None])
Creates a new todo list. `name` sets the name of the todo list (Unicode). Returns the ID of the created todo list.

vcalendar_string export_vcalendars(tuple<int,..>)
Returns a `vcalendar` string that contains the specified entries in vCalendar format. The parameter for this function is a tuple that contains the entry IDs of the exported entries.

entry_ids import_vcalendars(string)
Imports `vcalendar` entries, given in the string parameter, to the database. Returns a tuple that contains the unique IDs of the imported entries.

todo_lists
Enables the `TodoListDict` (dictionary-like) object to access the todo lists of this database.

del db[id]
Deletes the given calendar `Entry` from the database. `db` is the `CalendarDb` object and `id` is the unique ID of the calendar `Entry`.

EntryObject db[id]
Returns a calendar `Entry` object indicated by the unique ID. The returned object can be one of the following: `AppointmentEntry`, `EventEntry`, `AnniversaryEntry`, or `TodoEntry`. `db` is the `CalendarDb` object and `id` is the unique ID of the calendar `Entry`.

Success compact()
Compacts the database file. The returned value (integer) indicates the success of compaction; a value other than zero means that the compaction was successful.

16.3 Entry Objects

An `Entry` object represents a live view into the state of a single entry in the database. You can access the entries with an entry's unique ID. If you create a new entry using `db.add_appointment` etc., it is saved into the database only if you call the entry's `commit` method. In case an entry is already saved into the database, the autocommit mode is on by default and all the changes are automatically saved into the database, unless you call the entry's `begin` method. If you call the entry's `begin` method, the changes are not saved into the database until you call the entry's `commit` method.

Database entries cannot be locked. In other words, other applications are able to make changes to the database entries you are using (not directly to the `EntryObjects` you are using, but to their representation in the database) at the same time you are modifying them, even if you use `begin` and `commit` methods.

`Entry` objects have the following methods and properties:

`content`
: Sets or returns the entry's content text (Unicode).

`commit()`
: Saves the entry or in case of a new entry adds the entry into the database. Note that this can be called only in case of a new entry, created with `db.add_appointment` etc., or after `begin` is called.

`rollback()`
: Undoes the changes made after last `commit`.

`set_repeat(`*`dictionary`*`)`
: Sets the repeat data of the entry. *`dictionary`* is a repeat data dictionary that contains all the repeat rules. For more information on repeat rules, see Section *16.4*, *Repeat Rules*.

`repeat_dict` `get_repeat()`
: Returns the repeat data dictionary of the entry.

`location`
: Sets or returns the entry's location data (Unicode), for example meeting room information.

`set_time(`*`start, end`*`)`
: Sets the start and end datetime values of the entry (floats). If only one parameter is given, the other will have the same value.
In case of events, anniversaries, and todo entries the datetime values are truncated to corresponding date values.
`TodoEntries` can be made undated with `TodoEntry.set_time(None)`. Making the todo entry undated means removing the start and end date and all the repeat rules.

`start_time`
: Returns the start datetime value (float) of the entry or `None` if the start datetime of the entry is not set.

`end_time`
: Returns the end datetime value (float) of the entry or `None` if the end datetime of the entry is not set.

`id`
: Returns the unique ID of the entry.

`last_modified`
: Returns the datetime value (float) of the entry's last modification in universal time.

`alarm`
: Sets or returns an alarm datetime value (float) for the entry. Returns `None` if `alarm` is not set. Alternatively removes the alarm if the value is set to `None`.
Alarms can be set to all `Entry` types. However, only alarms set to Appointments and Anniversaries will actually cause an alarm; this is similar to the Calendar application in your Nokia device, which allows you to set an alarm only for Meetings and Anniversaries. In addition, alarms set to any entries residing in a database other than the default database do not cause actual alarms either.

`priority`
: Sets or returns the priority of the entry, which can be an integer ranging from 0 to 255. Native Phonebook and Calendar applications in Nokia devices use value 1 for high priority, 2 for normal priority, and 3 for low priority.

`crossed_out`
: Sets or returns the crossed out value of an entry. A value that is interpreted as false means that the entry is not crossed out, whereas a value that is interpreted as true means that the entry is crossed out. Note that `TodoEntries` must also have a cross-out time while the other entry types cannot have one. If `TodoEntry` is crossed out using this method, the moment of crossing out is set to the cross-out time of the `TodoEntry`. See also Section *16.3.4*, *TodoEntry*, `cross_out_time`.

`replication`
: Sets or returns the entry's replication status, which can be one of the following: `'open'`, `'private'`, or `'restricted'`.

`vcalendar_string` `as_vcalendar()`
: Returns this entry as a vCalendar string.

16.3.1 AppointmentEntry Objects

`AppointmentEntry` class contains no additional methods compared to the `Entry` class from which it is derived.

16.3.2 EventEntry

`EventEntry` class contains no additional methods compared to the `Entry` class from which it is derived.

16.3.3 AnniversaryEntry

`AnniversaryEntry` class contains no additional methods compared to the `Entry` class from which it is derived.

16.3.4 TodoEntry

`TodoEntry` objects represent todo entry types. They have additional properties compared to the `Entry` class from which they are derived.

`TodoEntry` objects have the following additional properties:

`cross_out_time`
: Sets or returns the cross-out date value of the entry. The returned value can be `None` meaning that the entry is not crossed out, or the cross-out date (float). The set value must be date (float). Setting a cross-out time also crosses out the entry. See also Section *16.3*, *Entry Object*, `crossed_out`.

`todo_list`
: Sets or returns the ID of the todo list to which this entry belongs.

16.3.5 TodoListDict

`TodoListDict` objects are dictionary-like objects that enable accessesing todo lists.

`TodoListDict` objects have the following property:

`default_list`
: Returns the ID of the default todo list.

16.3.6 TodoList

`TodoList` objects are dictionary-like objects that enable accessesing todo lists. `TodoList` objects have the following properties:

`name`
: Sets or returns the name of the todo list as a Unicode string.

`id`
: Returns the ID of the todo list as an integer.

16.4 Repeat Rules

Repeat rules specify an entry's repeat status, that is, the recurrence of the entry. There are six repeat types:

- `daily`: repeated daily
- `weekly`: repeat on the specified days of the week, such as Monday and Wednesday, etc.
- `monthly_by_dates`: repeat monthly on the specified dates, such as the 15th and 17th day of the month
- `monthly_by_days`: repeat monthly on the specified days, such as the fourth Wednesday of the month, or the last Monday of the month
- `yearly_by_date`: repeat yearly on the specified date, such as December 24
- `yearly_by_day`: repeat yearly on the specified day, such as every third Tuesday of May

There are exceptions to repeat rules. For example, you can specify the datetime value (float) in such a way that the entry is not repeated on a specific day even if the repeat rule would specify otherwise.

You must set the start and end dates (floats) of the repeat. The end date can also be set to `None` to indicate that the repeating continues forever. You can set `interval` defining how often the repeat occurs, for example in a daily repeat: `1` means every day, `2` means every second day, etc. You can also set the `days` specifier which lets you explicitly specify the repeat days; for example in a weekly repeat you can set `"days":[0,2]` which sets the repeat to occur on Mondays and Wednesdays. If you do not set the `days` specifier, the repeat days are calculated automatically based on the start date.

You can modify repeat data by calling `rep_data = entry.get_repeat`, then making changes to `rep_data` dictionary, and then calling `entry.set_repeat(rep_data)`.

Repeating can be cancelled by calling `entry.set_repeat` with a parameter that is interpreted to be false, such as `entry.set_repeat(None)`.

Repeat definition examples:

```
repeat = {"type":"daily", #repeat type
          "exceptions":[exception_day, exception_day+2*24*60*60],
          #no appointment on those days
          "start":appt_start_date, #start of the repeat
          "end":appt_start_date+30*24*60*60, #end of the repeat
          "interval":1} #interval (1=every day, 2=every second day etc.)
```

```
repeat = {"type":"weekly", #repeat type
          "days":[0,1], #which days in a week (Monday, Tuesday)
          "exceptions":[exception_day], #no appointment on that day
          "start":appt_start_date, #start of the repeat
          "end":appt_start_date+30*24*60*60, #end of the repeat
          "interval":1}
          #interval (1=every week, 2=every second week etc.)


repeat = {"type":"monthly_by_days", #repeat type
          # appointments on second Tuesday and last Monday of the month
          "days":[{"week":1, "day":1},{"week":4, "day":0}],
          "exceptions":[exception_day], #no appointment on that day
          "start":appt_start_date, #start of the repeat
          "end":appt_start_date+30*24*60*60, #end of the repeat
          "interval":1}
          #interval (1=every month, 2=every second month etc.)


repeat = {"type":"monthly_by_dates", #repeat type
          "days":[0,15],
          # appointments on the 1st and 16th day of the month.
          "exceptions":[exception_day], #no appointment on that day
          "start":appt_start_date, #start of the repeat
          "end":appt_start_date+30*24*60*60, #end of the repeat
          "interval":1}
          #interval (1=every month, 2=every second month etc.)


repeat = {"type":"yearly_by_date", #repeat type
          "exceptions":[exception_day], #no appointment on that day
          "start":appt_start_date, #start of the repeat
          "end":appt_start_date+3*365*24*60*60, #end of the repeat
          "interval":1}
          #interval (1=every year, 2=every second year etc.)


repeat = {"type":"yearly_by_day", #repeat type
          # appointments on the second Tuesday of February
          "days":{"day":1, "week":1, "month":1},
          "exceptions":[exception_day], #no appointment on that day
          "start":appt_start_date, #start of the repeat
          "end":appt_start_date+3*365*24*60*60, #end of the repeat
          "interval":1}
          #interval (1=every year, 2=every second year etc.)
```

17 contacts Module

The `contacts` module offers an API to address book services allowing the creation of contact information databases. The `contacts` module represents a Symbian contact database as a dictionary-like `ContactDb` object, which contains `Contact` objects and which is indexed using the unique IDs of those objects. A `Contact` object is itself a list-like object, which contains `ContactField` objects and which is indexed using the field indices. Unique IDs and field indices are integers. The `ContactDb` object supports a limited subset of dictionary functionality. Therefore, only `__iter__`, `__getitem__`, `__delitem__`, `__len__`, `keys`, `values`, and `items` are included.

`ContactDb` objects represent a live view into the database. If a contact is changed outside your Python application, the changes are visible immediately, and conversely any changes you commit into the database are visible immediately to other applications. It is possible to lock a contact for editing, which will prevent other applications from modifying the contact for as long as the lock is held. This can be done in, for example, a contacts editor application when a contact is opened for editing, very much like with the Contacts application in your Nokia device. If you try to modify a contact without locking it for editing, the contact is automatically locked before the modification and released immediately afterwards.

17.1 Module Level Functions

The following free functions – functions that do not belong to any class – are defined in the `Contact` module:

`ContactDb` `open([`*`filename`* `[,`*`mode`*`]])`

Opens a contacts database. *`filename`* should be a full Unicode path name. If *`filename`* is not given, opens the default contacts database. If *`mode`* is not given, the database must exist. If *`mode`* is '`c`', the database is created if it does not already exist. If *`mode`* is '`n`', a new, empty database is created, overwriting the possible previous database.

Warning: Using `open` together with the additional parameters *`filename`* or *`mode`* is intended for testing purposes only. Due to Series 60 SDK functionality, the `open` method can sometimes be unreliable with these parameters.

17.2 ContactDb Object

There is one default contact database, but it is possible to create several databases with the `open` function.

`ContactDb` objects have the following methods:

`Contact` `add_contact()`

Adds a new contact into the database. Returns a `Contact` object that represents the new contact. The returned object is already locked for modification. Note that a newly created contact will contain some empty default fields. If you do not want to use the default fields for anything, you can ignore them.

`matches` `find(`*`searchterm`*`)`

Finds the contacts that contain the given Unicode string as a substring and returns them as a list.

`import_vcards(`*`vcards`*`)`

Imports the vCard(s) in the given string into the database.

vcards `export_vcards((`*id*`,...))`
Converts the contacts corresponding to the given IDs to vCards and returns them as a string.

id_list `keys()`
Returns a list of unique IDs of all `Contact` objects in the database.

status `compact_required()`
Verifies whether compacting is recommended. Returns an integer value indicating either a true or false state. Returns `True` if more than 32K of space is unused and if this comprises more than 50 percent of the database file, or if more than 256K is wasted in the database file.

`compact()`
Compacts the database to its minimum size.

`del db[id]`
Deletes the given contact from the database.

typedict_list `field_types()`
Returns a list of dictionary objects that contains information on all supported field types. The list contains dictionary objects, which each describe one field type. The most important keys in the dictionary are `'type'` and `'location'` which together indentify the field type. `'type'` can have string values such as `'email_address'`. `'location'` can have the string values `'none'`, `'home'`, or `'work'`. Another important key is `'storagetype'`, which defines the storage type of the field. `'storagetype'` can have the string values `'text'`, `'datetime'`, `'item_id'`, or `'binary'`. Note that the `Contacts` extension does not support adding, reading, or modifying fields of any other type than `'text'` or `'datetime'`. The other content returned by `field_types` is considered to be advanced knowledge and is not documented here.

17.3 Contact Object

A `Contact` object represents a live view into the state of a single contact in the database. You can access the fields either with a contact's numeric field ID as `contact[fieldid]`, or using the `find` method. Attempting to modify a contact while it has been locked for editing in another application will raise the exception `ContactBusy`.

`Contact` objects have the following attributes:

`id`
The unique ID of this `Contact`. Read-only.

`title`
The title of this `Contact`. Read-only.

`Contact` objects have the following methods:

`begin()`
Locks the contact for editing. This prevents other applications from modifying the contact for as long as the lock is held. This method will raise the exception `ContactBusy` if the contact has already been locked.

`commit()`
Releases the lock and commits the changes made into the database.

`rollback()`
Releases the lock and discards all changes that were made. The contact remains in the state it was before `begin`.

vcard `as_vcard()`
Returns the contact as a string in vCard format.

`add_field(type[, value[, label=field_label][, location=location_spec])`
Adds a new field into this `Contact`. This method raises `ContactBusy` if the contact has been locked by some other application. `type` can be one of the supported field types as a string. The following field types can be added at present:

- `city`
- `company_name`
- `country`
- `date`
- `dtmf_string`
- `email_address`
- `extended_address`
- `fax_number`
- `first_name`
- `job_title`
- `last_name`
- `mobile_number`
- `note`
- `pager_number`
- `phone_number`
- `po_box`
- `postal_address`
- `postal_code`
- `state`
- `street_address`
- `url`
- `video_number`
- `wvid`

The following field types are recognized but cannot be created at present:

- `first_name_reading`
- `last_name_reading`
- `picture`
- `speeddial`
- `thumbnail_image`
- `voicetag`

All supported field types are passed as strings or Unicode strings, except for `'date'` which is a float that represents Unix time. For more information on Unix time, see Section *4.5*, *Date and Time*.

`field_label` is the name of the field shown to the user. If you do not pass a label, the default label for the field type is used.

`location_spec`, if given, must be `'home'` or `'work'`. Note that not all combinations of type and location are valid. The settings of the current contacts database in use determine which ones are valid.

`matching_fields` `find([type=field_type][, location=field_location])`
Finds the fields of this contact that match the given search specifications. If no parameters are given, all fields are returned.

`del contact[`*`fieldindex`*`]`
Deletes the given field from this contact. Note that since this will change the indices of all fields that appear after this field in the contact, and since the `ContactField` objects refer to the fields by index, old `ContactField` objects that refer to fields after the deleted field will refer to different fields after this operation.

17.4 ContactField Object

A `ContactField` represents a field of a `Contact` at a certain index. A `ContactField` has attributes, some of which can be modified. If the parent `Contact` has not been locked for editing, modifications are committed immediately to the database. If the parent `Contact` has been locked, the changes are committed only when `commit` is called on the `Contact`.

`ContactField` objects have the following attributes:

`label`
The user-visible label of this field. Read-write.

`value`
The value of this field. Read-write.

`type`
The type of this field. Read-only.

`location`
The location of this field. This can be `'none'`, `'work'`, or `'home'`.

`schema`
A dictionary that contains some properties of this field. The contents of this dictionary correspond to those returned by the `ContactDb` method `field_types`. The contents of this dictionary correspond to those returned by the `ContactDb` method `field_types`.

18 Extensions to Standard Library Modules

The following standard modules have been extended.

18.1 thread Module

The following function has been added to the standard `thread` module:

`ao_waittid(`*`thread_id`*`)`
: Synchronizes with the end of the execution of the thread identified by the given *`thread_id`*. The implementation is based on a Symbian OS active object. For the blocking behavior, see Section *7.2, Ao_lock Type*.

18.2 socket Module

Bluetooth (BT) support has been added to the standard `socket` module. The following related constants and functions are defined:

> **Note:** In release 1.0 the functions `bt_advertise_service`, `bt_obex_receive`, and `bt_rfcomm_get_available_server_channel` incorrectly expected to be given the internal `e32socket.socket` object as the socket parameter instead of the proper `socket` object. Now the functions work correctly. The old calling convention is still supported but it is deprecated and may be removed in a future release.

`AF_BT`
: Represents the Bluetooth address family.

`BTPROTO_RFCOMM`
: This constant represents the Bluetooth protocol RFCOMM.

`RFCOMM, OBEX`
: Bluetooth service classes supported by `bt_advertise_service`.

`AUTH, ENCRYPT, AUTHOR`
: Bluetooth security mode flags.

`bt_advertise_service(`*`name, socket, flag, class`*`)`
: Sets a service advertising the service *`name`* (Unicode) on local channel that is bound to *`socket`*. If *`flag`* is `True`, the advertising is turned on, otherwise it is turned off. The service class to be advertised is either RFCOMM or OBEX.

`(`*`address, services`*`) bt_discover([`*`address`*`])`
: Performs the Bluetooth device discovery (if the optional BT device address is not given) and the discovery of RFCOMM class services on the chosen device. Returns a pair: BT device address, dictionary of services, where Unicode service name is the key and the corresponding port is the value.

`(`*`address, services`*`) bt_obex_discover([`*`address`*`])`
: Same as `discover`, but for discovery of OBEX class services on the chosen device.

`bt_obex_send_file(`*`address, channel, filename`*`)`
: Sends file *`filename`* (Unicode) wrapped into an OBEX object to remote *`address`*, *`channel`*.

`bt_obex_receive(`*`socket, filename`*`)`
: Receives a file as an OBEX object, unwraps and stores it into *`filename`* (Unicode). *`socket`* is a bound `OBEX` socket.

port `bt_rfcomm_get_available_server_channel(`*socket*`)`
Gets an available RFCOMM server channel for *socket*.

`set_security(`*socket*`, `*mode*`)`
Sets the security level of the given bound *socket*. The *mode* is an integer flag that is formed using a binary `or` operation of one or more of: `AUTH` (authentication), `ENCRYPT`, `AUTHOR` (authorization). Example: `set_security(s, AUTH | AUTHOR)`.

Note: When listening to a Bluetooth socket on the phone, it is necessary to set the security level.

Note: SSL is not supported in Series 60 1st Edition. SSL client certificates are not supported at all.

For examples on the usage of these functions, see *Programming with Python for Series 60 Platform [6]*.

19 Terms and Abbreviations

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

AAC; Adaptive Audio Coding	AAC provides basically the same sound quality as MP3 while using a smaller bit rate. AAC is mainly used to compress music.
Advertise	Advertise service in Bluetooth makes it known that a certain Bluetooth service is available.
AMR	Adaptive Multi-rate Codec file format.
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.
BPP	Bits Per Pixel
C STDLIB	Symbian OS's implementation of the C standard 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 finds other nearby Bluetooth devices and their advertised services.
DLL	Dynamic link library
GSM; Global System for Mobile communication	GSM is a digital mobile telephone system that uses a variation of time division multiple access. It digitizes and compresses data, then sends it down a channel with two other streams of user data, each in its own time slot.
GUI	Graphical User Interface
I/O	input/output
IP	Internet Protocol
MBM; MultiBitMap	The native Symbian OS format used for pictures. MBM files can be generated with the "`bmconv.exe`" tool included in the Series 60 SDK.
MIDI; Musical Instrument Digital Interface	A protocol and a set of commands for storing and transmitting information about music.
MIF; Multi-Image File	MIF files are similar to MBM files and can contain compressed SVG-T files. This file type can be generated with the "`MifConv.exe`" tool.
MIME; Multipurpose Internet Mail Extensions	MIME is an extension of the original Internet e-mail protocol that can be used to exchange different kinds of data files on the Internet.
MP3	A standard technology and format for compressing a sound sequence into a very small file while preserving the original level of sound quality when it is played.
OS	Operating System
Real Audio	An audio format developed by Real Networks.

RDBMS	Relational database management system
SMS; Short Message System (within GSM)	SMS is a service for sending messages of up to 160 characters, or 224 characters if using a 5-bit mode, to mobile phones that use GSM communication.
Softkey	Softkey is a key that does not have a fixed function nor a function label printed on it. On a phone, selection keys reside below or above on the side of the screen, and derive their meaning from what is presently on the screen.
SQL	Structured Query Language
SVG, SVG-T; Scalable Vector Graphics (-Tiny)	XML-based vector graphics format for describing two-dimensional graphics and graphical applications.
Twip	Twips are screen-independent units to ensure that the proportion of screen elements are the same on all display systems. A twip is defined as 1/1440 of an inch, or 1/567 of a centimeter.
UI	User Interface
UI control	UI control is a GUI component that enables user interaction and represents properties or operations of an object.
WAV	A file format for recording sound, especially in multimedia applications.

20 References

1. G. van Rossum, and F.L. Drake, Jr., editor. [Python] Library Reference. Available at http://www.python.org/doc
2. G. van Rossum, and F.L. Drake, Jr., editor. Extending and Embedding [the Python Interpreter]. Available at http://www.python.org/doc
3. G. van Rossum, and F.L. Drake, Jr., editor. Python/C API [Reference Manual]. Available at http://www.python.org/doc
4. Series 60 SDK documentation
5. Getting Started with Python for Series 60 Platform

   http://www.forum.nokia.com/
6. Programming with Python for Series 60 Platform

   http://www.forum.nokia.com/
7. *Audio & Video* section on the *Forum Nokia* Web site (for Nokia devices)

   http://www.forum.nokia.com/audiovideo
8. *Developers* section on the *Series 60 Platform* Web site (for all Series 60 devices)

   http://www.series60.com/
9. Python for Series 60 developer discussion board
   http://discussion.forum.nokia.com/
10. Scalable Vector Graphics (SVG) 1.1 Specification
    http://www.w3.org/TR/SVG/

Appendix A Python Library Module Support

Table 2: Status of library module support

Name	Type	Status	Remarks
`_testcapi`	PYD	Y	
`anydbm`	PY	X	DBM API is implemented by PY `e32dbm` that relies on PYD `e32db` (see Chapter *9*, *e32dbm Module*)
`atexit`	PY	X	
`base64`	PY	X	
`bdb`	PY	(X)	
`binascii`	built-in	X	
`cmd`	PY	(X)	
`code`	PY	X	
`codecs`	PY	X	
`codeop`	PY	X	
`copy`	PY	X	
`copy_reg`	PY	X	
`cStringIO`	built-in	X	
`dis`	PY	(X)	
`errno`	built-in	X	
`exceptions`	built-in	X	
`__future__`	PY	X	
`httplib`	PY	X	
`imp`	built-in	X	
`keyword`	PY	X	
`linecache`	PY	X	
`marshal`	built-in	X	
`math`	built-in	X	
`md5`[1]	built-in	X	
`mimetools`	PY	X	
`operator`	built-in	X	
`os`, `os.path`	PY	X	Wraps built-in `e32posix`. Limitations discussed in Section *4.9*, *Limitations and Areas of Development*.
`pdb`	PY	(X)	
`quopri`	PY	X	

[1] Derived from the RSA Data Security, Inc. MD5 Message-Digest Algorithm

Name	Type	Status	Remarks
`random`	PY	X	
`re`	PY	X	Uses PY `sre` as its engine.
`repr`	PY	X	
`rfc822`	PY	X	
`select`	PY	X	A minimal implementation: `select` is supported only for input from sockets.
`socket`	PY	X	Requires PYD `e32socket`. Contains extensions as described in Section *18.2*, *socket Module*. Limitations discussed in Section *4.9*, *Limitations and Areas of Development*.
`sre`	PY	X	Wraps built-in `_sre`.
`string`	PY	X	
`StringIO`	PY	X	
`struct`	built-in	X	
`sys`	built-in	X	
`thread`	built-in	X	Contains extensions as described in Section *18.1*, *thread Module*
`threading`	PY	(X)	
`time`	built-in	X	
`traceback`	PY	X	
`types`	PY	X	
`urllib`	PY	X	
`urlparse` (urlsplit only)	PY	X	
`uu`	PY	X	
`warnings`	PY	X	
`whichdb`	PY	X	
`xreadlines`	built-in	X	
`zipfile`	PY	X	
`zlib`	PYD	X	

Table 2 uses the following coding for module types:

- PY – module is implemented in Python.
- Built-in – module is a built-in C/C++ module.
- PYD – module is a dynamically loadable C/C++ module.

For support status, the following codes are used:

- X – included to the Series 60 Python distribution.
- (X) – not included to the Series 60 Python distribution, but works both on phone and SDK.

- Y – included only to the SDK distribution.

Appendix B Extensions to C API

The native API exported by the interpreter in Series 60 environment consists of `class CSPyInterpreter`, Python/C API (see *G. van Rossum, and F.L. Drake, Jr., editor. Python/C API [Reference Manual]. [3]*), and a small set of extensions to Python/C API.

B.1 class CSPyInterpreter

`class CSPyInterpreter` offers an interface for initializing the interpreter and for running scripts. It exports the following public interface:

```
static CSPyInterpreter*
NewInterpreterL(TBool aCloseStdlib = ETrue,
                void(*aStdioInitFunc)(void*) = NULL,
                void* aStdioInitCookie = NULL);
TInt RunScript(int argc, char** argv);
void PrintError();
void (*iStdI)(char* buf, int n);
void (*iStdO)(const char* buf, int n);
```

The caller of the constructor `CSPyInterpreter::NewInterpreterL` may provide its own function `aStdioInitFunc` for initializing Symbian OS STDLIB's standard I/O descriptors. It gets called with the argument `aStdioInitCookie`. The `CSPyInterpreter` class can also be requested to leave STDLIB open at its destruction.

The `CSPyInterpreter::RunScript` method establishes a Python interpreter context and runs the script file whose full path name is in `argv[0]` with the given argument vector. After completion, it leaves the interpreter context and returns a Symbian error code to indicate success or failure.

The `CSPyInterpreter::PrintError` method can be used to print current Python exception information to the standard error output.

B.2 Extensions to C API

Defined in `symbian_python_ext_util.h`:

`PyObject* SPyErr_SetFromSymbianOSErr(int error)`
: Sets Python exception of type `PyExc_SymbianError` with the value field set to symbolic name of the Symbian OS enumeration value `error` and returns `NULL`. In case `error` has the special value `KErrPython`, it assumes that a Python exception has already been set and returns `NULL`.

`SPyAddGlobal()`, `SPyAddGlobalString()`, `SPyGetGlobal()`, `SPyGetGlobalString()`, `SPyRemoveGlobal()` and `SPyRemoveGlobalString()`
: Can be used for storing module implementation's global data. They are thin wrappers around `PyDict_SetItem`, `PyDict_SetItemString`, `PyDict_GetItem`, `PyDict_GetItemString`, `PyDict_DelItem`, `PyDict_DelItemString`, correspondingly, and can be used in the same way.

`PYTHON_TLS->thread_state` (defined in `"python_globals.h"`)
: Thread state and interpreter lock management must be performed according to the instructions; see *G. van Rossum, and F.L. Drake, Jr., editor. Python/C API [Reference Manual]. [3]*. Python for Series 60 Platform extends the Python/C API by offering a facility for querying the related Python thread state (`"PYTHON_TLS -> thread_state"`) from the context of the currently running thread. This can be used to re-establish the interpreter context with `PyEval_RestoreThread` in C/C++ code.

To Save/restore the interpreter context:

```
Py_BEGIN_ALLOW_THREADS
[...your code...]
Py_END_ALLOW_THREADS
```

To Restore/save the interpreter context:

```
PyEval_RestoreThread(PYTHON_TLS->thread_state)
[... your code...]
PyEval_SaveThread()
```

Defined in `pythread.h` (extends the standard `thread` module's C API):

`PyThread_AtExit(void(*)())`

Can be used for registering thread-specific exit functions. In the main thread calling this function has the same effect as calling `Py_AtExit`. For more information, see *G. van Rossum, and F.L. Drake, Jr., editor. [Python] Library Reference. [3].*

Appendix C Extending Series 60 Python

C.1 Overview

The general rules and guidelines for writing Python extensions apply in the Series 60 Python environment as well; for more information, *see G. van Rossum, and F.L. Drake, Jr., editor. Extending and Embedding [the Python Interpreter].[2]*. The Python/C API is available, see *G. van Rossum, and F.L. Drake, Jr., editor. Python/C API [Reference Manual]. [3]*. In addition, for an example on porting a simple extension to Series 60, see *Programming with Python for Series 60 Platform [6*].

The issues that need to be considered in the implementation of the extension modules include:

- Preparation of the data structures that make the C/C++ coded extensions visible to the Python interpreter and make it possible to perform calls from Python to C/C++ code
- Conversions between C/C++ representations of the Python objects and object types used in the extension code
- Maintenance of the reference counts of the C/C++ representations of the Python objects
- Passing of exceptions between C/C++ code and Python
- Management of interpreter's thread state and the interpreter lock

In addition to the concerns common for all Python C extensions, the following principles should be considered when implementing new Python interfaces in the Series 60 environment:

- Maximize the usage of Python's built-in types at the interfaces.
- Related to the above: design interfaces in such a way that information can be passed between them with minimal conversions.
- Convert Symbian operating system exceptions / errors to Python exceptions.
- Unicode strings are used at the interfaces to represent text that gets shown on the GUI. They can be passed to and from Symbian operating system without conversions.
- While performing potentially long-lasting / blocking calls from an extension implementation to services outside the interpreter, the interpreter lock must be released and then re-acquired after the call.
- Rather than always implementing a thin wrapper on top of a Symbian OS facility, consider the actual task for which the script writer needs the particular interface. For example, if the task involves interaction with the users using the GUI, the script writer's interest may well be limited to performing the interaction / information exchange in a way that is compatible with the UI style rather than having full control of the low-level details of the GUI implementation.
- The C/C++ implementation of a Python interface should be optimized for performance and covering access to the necessary features of the underlying Platform. Where necessary, the Python programming interface can be further refined by wrapper modules written in Python.

An extension module is packaged in its own dynamically loadable library that must be installed into `\system\libs` directory and named `<module_name>.pyd`. The module initialization function must be exported at ordinal 1. The module identification is based on the filename only. As a special feature of Series 60 Python, an optional module finalizer function may be exported at ordinal 2.

Warning: The macro versions of memory-management functions `PyMem_MALLOC` and `PyObject_NEW` are not included. Use the functions `PyMem_Malloc` and `PyObject_New` instead.

C.2 Services for Extensions

Series 60 Python Platform implements an adaptation layer between Series 60 UI application framework and script language UI extensions to simplify UI extension development. This API is used by the implementation of the `appuifw` module but not exported in the current release. Some general utility services for extensions are also provided, see *Appendix B*, *Extensions to C API*.

C.3 Example

The extension code snippet in Figure 9 demonstrates some of the issues mentioned in *C.1*, *Overview*:

```
01 extern "C" PyObject *
02 note(PyObject* /*self*/, PyObject *args)
03 {
04   TInt error = KErrNone;
05   int l_tx, l_ty;
06   char *b_tx, *b_ty;
07
08   if (!PyArg_ParseTuple(args, "u#s#", &b_tx, &l_tx, &b_ty, &l_ty))
09     return NULL;
10
11   TPtrC8 stype((TUint8*)b_ty, l_ty);
12   TPtrC note_text((TUint16 *)b_tx, l_tx);
13   CAknResourceNoteDialog* dlg = NULL;
14
15   if (stype.Compare(KErrorNoteType) == 0)
16     dlg = new CAknErrorNote(ETrue);
17   else if (stype.Compare(KInfoNoteType) == 0)
18     dlg = new CAknInformationNote(ETrue);
19   else if (stype.Compare(KConfNoteType) == 0)
20     dlg = new CAknConfirmationNote(ETrue);
21   else {
22     PyErr_BadArgument();
23     return NULL;
24   }
25
26   if (dlg == NULL)
27     return PyErr_NoMemory();
28
29   Py_BEGIN_ALLOW_THREADS
30   TRAP(error, dlg->ExecuteLD(note_text));
31   Py_END_ALLOW_THREADS
32
33   if (error != KErrNone)
34     return SPyErr_SetFromSymbianOSErr(error);
35   else {
36     Py_INCREF(Py_None);
37     return Py_None;
38   }
39 }
```

Figure 9: An example C/C++ extension snippet.

- Conversion from Python data types, usage of built-in data types at extension interface, usage of Unicode strings (lines 8-12)
- Maintenance of the reference counts (line 36)
- Passing of exceptions between C/C++ code and Python (line 34)

- Releasing the interpreter lock while performing a blocking call to a service outside the interpreter (lines 29, 31)
- Simplifying the API to the note facility of the Platform