Reviewer Approved
Featured Article

Contents

- 1 Introduction
- 2 Development tools
- 3 Basic approach
- 4 Methods and functions
    - 4.1 File functions
    - 4.2 Feature functions
    - 4.3 Application operations
- 5 Part 1
    - 5.1 save_location()
    - 5.2 load_location()
    - 5.3 save_reminder()
    - 5.4 load_reminder()
    - 5.5 dictionary_copy(dictionary)
    - 5.6 location_list()
- 6 Screenshots
- 7 See also

Introduction

Do you want to develop location-based applications?

This article is the first in a series approaching all the basics for developing a location-based application for mobile devices. The application can be defined as a Geo-Scheduler or a Location scheduler. The simplest rapid mobile application development tool, PyS60, is used to develop the application.

The following modules are used when developing this application:

- appuifw module
- audio module
- appswitch module (extension)
- e32 module

- time module
- os module
- location module
- envy module (extension)

The extensions listed above can be downloaded from the C++ Python Extensions.

Development tools

1) A text editor

(Preferred editor: Python's IDLE - Integrated DeveLopment Environment)

2) Python for S60

(Preferred version: latest release 1.4.4)

3) Python for S60 script shell

(Preferred version: Latest release 1.4.4)

4) Extensions listed above which are not bundled with PyS60

The PyS60 tools can be downloaded from Sourceforge resources.

Basic approach

In this application, localization is done using the network information. The network information can be provided using the location module, *cellid*. If you are not familiar with retrieving the cellid using Pys60, using the appswitch module, or using the envy module, read the following articles before you proceed.

How to switch application in foreground

How to use the envy module

How to get info on cell location

Methods and functions

The following methods or functions are used:

File functions

1) `save_location`: Save location information into file

2) `load_location`: Read location information into variables

3) `save_reminder`: Save reminder information into a file

4) `load_reminder`: Read reminder information into variables

5) `dictionary_copy`: Dictionary copy

6) `location_list`: Return the location names

Feature functions

1) `add_location`: Add a new location

2) `edit_location`: Edit a location

3) `add_reminder`: Add a new reminder

4) `edit_reminder`: Editing screen function

5) `edit_reminder_bylocation`: Edit reminder by location function

6) `edit_reminder_byreminder`: Edit reminder by reminder function

Application operations

1) `ms_handler`: MS option handler

2) `main`: Start program

3) `tracker`: Main tracker function

4) `background_handler`: Background handler options

Part 1

This article - Part 1 - discusses the **File functions** in detail.

save_location()

This function adds the location information to the database file. A file handler named `file_handler` is created to handle file operations. A .dat file specified by the variable `file_handler_location` is accessed here. In this example, the `location.dat` file is in C:, and it is assigned to the `file_handler_location`.

Note: `location.dat` contains locations added by the `save_location()` function.

```
file_handler_location = "c:\\location.dat"
```

The geo and cellids are populated before running the file functions to NULL.

Note: The geo and cellids must be cleared by using the `clear()` function so that they do not contain any previously used values.

The definition for the `save_location()` function is as follows:

```
def save_location():
    file_handler = file(file_handler_location, "w")
    temporary = dictionary_copy(cellids)

    cellids.clear()
    geo.clear()

    for cell, loc in temporary.items():
        file_handler.write(str(cell) + ':' + str(loc) + '\n')
        cellids[cell] = loc
        geo[loc] = cell
    file_handler.close()
```

The function `dictionary_copy(cellids)` is defined here.

load_location()

This function is responsible for loading the locations into variables. Here the value of the cellid is stored into variables.

The same `file_handler` and the `file_handler_location` are used to access the `location.dat` this time to read the location values.

```
file_handler = file(file_handler_location, "r")
```

Using iterations, `location.dat` is read for locations/values of cellid. The values are loaded to variable geo[loc].

geo and cellids variables are populated before calling the load function.

```
cellids = {}

geo = {}
```

The definition for the `load_location()` function is as follows:

```
def load_location():
    file_handler = file(file_handler_location, "r")
    for line in file_handler:
        cell, loc = line.split(":")
        cell = cell.strip()
        loc = loc.strip()
        cellids[cell] = loc
        geo[loc] = cell
    file_handler.close()
```

save_reminder()

The `save_reminder()` function adds a new reminder to the file.

The `file_handler_reminder` is referenced in the .dat file used to record reminders.

In the following example, the `reminder.dat` file is in C:, and it is assigned to the variable `file_handler_reminder`.

```
file_handler_reminder = "c:\\reminder.dat"
```

`dictionary_copy(items)` is defined here. It returns the value of the temporary variable which was initialized as NULL in the scope of the `dictionary_copy(dictionary)` function.

The reminder string and the value of the cell are written to the file `reminder.dat`.

The definition for the `save_reminder()` function is as follows:

```
def save_reminder():
    file_handler = file(file_handler_reminder, "w")
    temporary = dictionary_copy(rems)
    rems.clear()
    for i, value in enumerate(temporary.values()):
```

```
        file_handler.write(str(i) + ':' + str(value['cell']) + ':' + str(value['date']) + ':' + v
        rems[str(i)] = {'cell':value['cell'], 'date':value['date'], 'desc':value['desc']}
    file_handler.close()
```

load_reminder()

This function reads the reminders in the `file_handler_reminder`, that is, from the `reminder.dat` file.

`file_handler` is used to open the `reminder.dat` to read values.

The values of the </tt>reminder.dat `(description) are loaded into the variable <tt>rems` (reminders).

```
rems[id] = {'cell':cell, 'date':str(date), 'desc':desc}
```

Finally, the definition for `load_reminder()` is as follows:

```
def load_reminder():
    temporary = {}
    file_handler = file(file_handler_reminder, "r")
    for line in file_handler:
        id, cell, date, desc = line.split(":")
        id = id.strip()
        cell = cell.strip()
        date = date.strip()
        desc = desc.strip()
        rems[id] = {'cell':cell, 'date':str(date), 'desc':desc}
    file_handler.close()
```

dictionary_copy(dictionary)

In `save_location` and `save_reminder`, the function returns the value of the temporary variable.

A parameter must be passed to the dictionary function.

For example, in the `ave_location()` function it is used as follows:

```
temporary = dictionary_copy(cellids)
```

The definition of `dictionary_copy()` is:

```
def dictionary_copy(dictionary):
    return dict(dictionary)
```

location_list()

`location_list()`: Return the location names

There is no need to pass a parameter to this function. However, the variable `cellids` can always be passed for its global. However, the preferred way is to have it without the parameters.

In the same way as the function `dictionary_copy(dictionary)`, this also returns the temporary variable, but from the items of the `cellids` variable.

The definition for the `location_list()` function is:

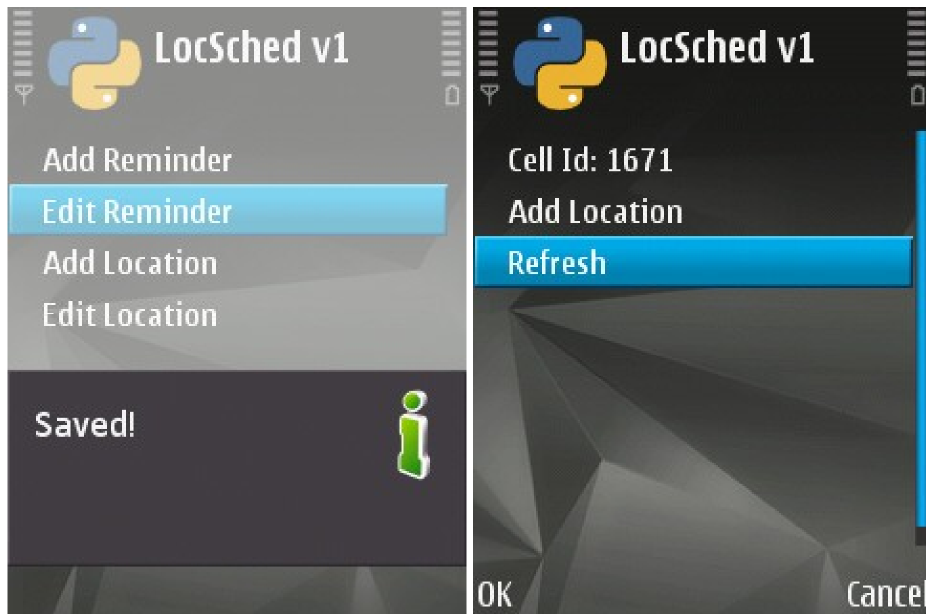
```
def location_list():
    return [unicode(loc) for loc in cellids.values()]
```

That finishes the file functions (Part 1) for the Geo-scheduler.

These functions are called when the user selects an action from a custom menu.

Screenshots

These screenshots demonstrate the menu and how these functions are be used. The screenshots are relevant only to Part 1.



See also

- How do develop a Geo-scheduler application - Part 2
- How do develop a Geo-scheduler application - Part 3
- PyS60